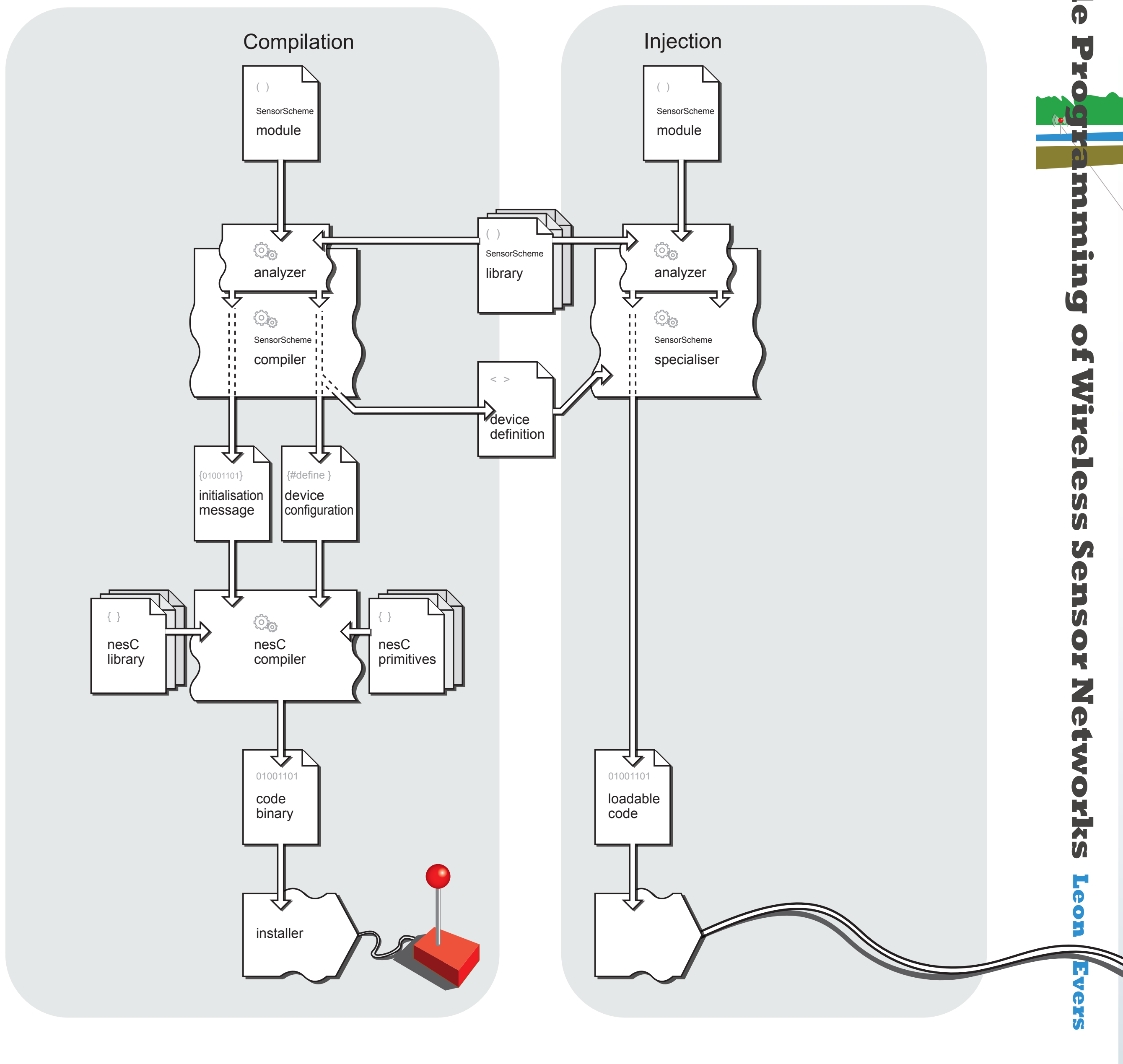
As computers get smaller, their use is becoming more widespread and ubiquitous and soon they will be all around us. They will be sensing their environment as part of wireless sensor networks and communicating their findings to us. These nodes are battery-operated and need to be small and cheap to be economically viable, and have only little CPU power and memory available...

This dissertation describes and motivates the design and implementation of the SensorScheme platform, a programming language and interpreter designed to make available proper methods and abstractions to program wireless sensor networks effectively, safely transport new programs to running networks, and reduce the size of programs.

ISBN 978-90-365-3028-6



Concise and Flexible Programming of Wireless Sensor Networks Leon Evers

Concise and Flexible Programming of Wireless Sensor Networks

Leon Evers

Concise and Flexible Programming of Wireless Sensor Networks

Leon Evers

Keywords: Wireless Sensor Networks, Programming Languages, Reprogramming, Wireless communication.

Cover design: Sonja Evers

Printed by Wöhrmann Print Service.

ISBN 978-90-365-3028-6

CONCISE AND FLEXIBLE PROGRAMMING OF WIRELESS SENSOR NETWORKS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. H. Brinksma,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op donderdag 24 juni 2010 om 13.15 uur

door

Leon Evers

geboren op 27 mei 1979

te Apeldoorn.

Dit proefschrift is goedgekeurd door

Prof.	Dr.	P.J.M	Havinga	(promotor)
	Dr.	J.	Kuper	(assistent-promotor)

Preface

Any sufficiently complicated C or Fortran program contains an ad hoc, informally-specified, bug-ridden, slow implementation of half of Common Lisp.

Greenspun's Tenth Rule of Programming

The document in front of you is the culmination of almost five years of work in which I have been attempting to gain knowledge on the relatively new field of wireless sensor networks and contribute some of my new insights back to the scientific community. It has been an interesting and exciting journey which has taught me a lot about a great variety of subjects, stretching large parts of computer science research, from communication protocol design to distributed database systems, from CPU design to the theory of programming languages.

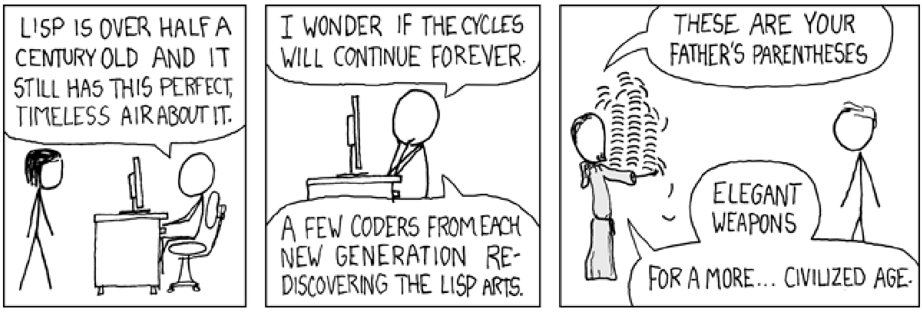
Before we address the content of this dissertation I'd like to introduce the reader to the way in which it came about. After all, the motivations and intentions behind scientific discoveries do often explain and illustrate these discoveries in an interesting way.

At the start of this research I had already been getting myself familiar to the field of wireless sensor networks by way of course assignments and my Master's thesis. As I was looking for a topic to fill my doctoral thesis with I encountered over and over again a expressions of how constructing wireless sensor network programs was so much harder than 'usual' programs, something that completely matched my own experience. If sensor networks ever were to contribute to improving society and our quality of life, this might be the issue most in need of being addressed. As potential users of this technology, we all want reliable technology that doesn't surprise us with random crashes or lockups (as we more or less have come to expect of computers). As software developers, a pleasurable experience in creating the next great application of this new technology would certainly improve our life.

During these early days I stumbled upon the quote at the top of this preface, which started my investigations into the Lisp programming language, which I did not know. The essays by Paul Graham [1] and others on their experience with and opinions on this language certainly raised my curiosity. "If this language really is as great and powerful as they say, I need to know what is going on."

While I would like to refrain from 'educating' the community about the benefit of using 'Lispy' languages in this new field, it has been hard at times to keep from doing just that. And in a way, this dissertation is my attempt at showing what this approach can bring to wireless sensor network platforms.

[1] `http://www.paulgraham.com/articles.html`


LISP IS OVER HALF A CENTURY OLD AND IT STILL HAS THIS PERFECT, TIMELESS AIR ABOUT IT.
I WONDER IF THE CYCLES WILL CONTINUE FOREVER.
A FEW CODERS FROM EACH NEW GENERATION RE-DISCOVERING THE LISP ARTS.
THESE ARE YOUR FATHER'S PARENTHESES
ELEGANT WEAPONS
FOR A MORE... CIVILIZED AGE.
Picture obtained from the xkcd webcomic by Randall Munroe at xkcd.com

Abstract

As computers get smaller, their use is becoming more widespread and ubiquitous. Soon computers will be all around us, interacting with us in our daily lives as networks of *smart objects*, or sensing their environment and communicating its findings as *wireless sensor networks*.

These wireless sensor network devices, or *sensor nodes* as we call them, communicate with each other using low data rate digital radios. The devices are battery-operated and need to be small and cheap to be economically viable. Naturally, research in this field focuses on making efficient use of the scarce resources available, such as computation time and communication. The limited memory complicates implementation of many of the features common on 'larger' computers.

Writing applications on severely limited sensor network nodes is challenging for a number of reasons.

First, finding proper *methods* and *abstractions* with which to program wireless sensor networks effectively is an ongoing process. Small size of sensor network programs is crucial, to minimize development time, duration of transport and memory use. Powerful and effective abstractions and tools may reduce the size of programs and extend their capabilities.

Second, the challenge is to *program* devices with their desired behavior. For some applications only once, or infrequently, in other cases regular reprogramming is required. The low bandwidth communication and little memory make the transport of programs and their execution a non-trivial task. Furthermore, for wirelessly accessible and (re)programmable devices protection from loading malicious applications is also highly desirable.

This dissertation motivates and describes the design and implementation of the SensorScheme platform. SensorScheme is a programming language and interpreter designed to addresses the above challenges, in the following ways:

- SensorScheme makes use of a program interpreter to achieve platform independence and hardware protection, and to facilitate wireless (re)programming of sensor nodes. SensorScheme's interpreter is based not on a virtual machine design, common in the state of the art. Instead, SensorScheme's interpreter evaluates expressions, represented as trees of linked lists, inspired by its heritage of the Scheme programming language.

- SensorScheme is designed to use the state of the art in wireless sensor network programming techniques such as automatic memory management, multi-threading behavior, and the *reduction strategy* for combining sensor data from multiple nodes. SensorScheme advances the state of the art, making available techniques such as closures and continuations, and programming abstractions such as higher order functions. Furthermore, SensorScheme makes use of its property of *homoiconicity* – having the same representation of code and data, to wirelessly transport and dynamically load entire programs as well as single functions.

- The availability of this extended set of programming abstractions and techniques makes it possible to express sensor network applications concisely. This reduces the memory and communication bandwidth requirements for storing and transporting applications.

- SensorScheme introduces ObjectStreams, a communication technique using serialization to automatically encode and decode messages. ObjectStreams is integrated into the language and interpreter to facilitate the transport of programs into the network. It furthermore simplifies writing communicating programs by removing the program's dependency on the packet size of the underlying communication hardware.

- SensorScheme introduces a method of macro-programming networks of heterogeneous sensing and actuation devices. It simplifies construction and maintenance of these networks by writing a single network-wide program that includes the functionality of all nodes in the network. Specialization of this network-wide program (by partial evaluation) produces node-specific programs that include only the functionality required by this node, resulting in significant size reduction compared to the network-wide program.

We motivate and illustrate SensorScheme's design through the use of four application scenarios, two of which are extensively dealt with in the literature, while the other two present challenges of programmability unaddressed by the state of the art.

Using example implementations of each of the application scenarios this work presents the programming techniques and abstractions provided by SensorScheme, aimed at reducing the complexity of writing sensor network applications.

We also evaluate the SensorScheme platform using these example implementations. We show its suitability for wireless sensor network platforms in terms of communication, energy, computation and memory requirements for the presented applications. We furthermore assess SensorScheme's ease of programming measured as the program's sizes compared to the state of the art.

As is generally the case, a higher level of abstraction is to be traded off for an increase in resource consumption such as computation time, memory use, communication bandwidth, and ultimately energy.

Our results show that the computation overhead caused by interpretation is acceptable for typical WSN scenarios, causing no significant slowdown and only a minor increase in energy use.

Next, we show that while the ObjectStreams communication mechanism requires some communication protocols to be restructured, it offers comparable communication requirements for a similar level of application functionality.

Furthermore, we demonstrate that (interpreted) SensorScheme programs can be considerably shorter than their natively compiled counterparts. This makes it possible to provide a wider range of application functionality with the memory available to WSN nodes, while making it easier to write these programs.

In conclusion, even for resource-scarce wireless sensor networks the benefits for programmability of dynamic, high-level languages are within reach for wireless sensor networks, and offer levels of functionality not available in other ways.

Samenvatting

Omdat computers steeds kleiner worden, is het gebruik ervan steeds meer wijdverbreid en alomtegenwoordig. Binnenkort zullen computers overal om ons heen zijn, in wisselwerking met ons in ons dagelijks leven als netwerken van *slimme objecten*, of meten hun omgeving en communiceren hun bevindingen als *draadloze sensor netwerken.*

Deze draadloze sensor netwerk apparaten, of *sensor nodes* zoals wij ze noemen, communiceren met elkaar met behulp van lage datasnelheid digitale radio's. De apparaten werken op batterijen en moeten klein en goedkoop zijn om economisch levensvatbaar te zijn. Onderzoek op dit gebied richt zich op het maken van een efficiënt gebruik van de schaarse middelen die beschikbaar zijn, zoals de rekentijd en communicatie. De beperkte geheugenruimte bemoeilijkt de uitvoering van veel van de kenmerken die horen op 'grotere' computers.

Het schrijven toepassingen op zeer beperkte sensor netwerk nodes is een uitdaging om een aantal redenen.

Ten eerste, het vinden van een goede methoden en abstracties waarmee het programmeren van draadloze sensor netwerken effectief is, is een doorlopend proces. Het is van cruciaal belang dat sensor-netwerk programma's kort zijn, om de ontwikkeltijd, duur van het transport en het geheugengebruik te minimaliseren. Krachtige en effectieve gereedschappen kunnen vermindering van de omvang van de programma's en het uitbreiden van hun mogelijkheden realiseren.

Ten tweede, de uitdaging is om apparaten te *programmeren* met hun gewenste gedrag. Voor sommige toepassingen slechts een keer, of zelden, in andere gevallen is regelmatige herprogrammering nodig. De lage bandbreedte communicatie en weinige geheugen beschikbaar voor het transport van programma's en de uitvoering ervan maakt dit een niet-triviale taak. Bovendien, als draadloos toegankelijke en (her)programmeerbare apparaten is bescherming tegen het laden van kwaadaardige applicaties ook zeer wenselijk.

Dit proefschrift beschrijft en motiveert het ontwerp en de uitvoering van het SensorScheme platform. SensorScheme is een programmeertaal en executie-omgeving bedoeld om de bovengenoemde uitdagingen te adresseren, op de volgende manieren:

- SensorScheme maakt gebruik van een programma interpeter om platformonafhankelijkheid en hardware bescherming te bereiken, en om draadloos (her)programmeren van de sensor nodes te vergemakkelijken. De SensorScheme interpreter is niet gebaseerd op het ontwerp van een virtuele machine, wat gebruikelijk is in de huidige stand van de techniek. In plaats daarvan evalueert de SensorScheme interpreter uitdrukkingen, vertegenwoordigd als bomen van gelinkte lijsten, geĮnspireerd door haar erfgoed van de programmeertaal Scheme.

- SensorScheme is ontworpen om de stand van de techniek in de draadloze sensor netwerk programmeren technieken gebruiken, zoals automatisch geheugenbeheer, multi-threading gedrag, en de *reductie strategie* voor het combineren van sensorgegevens uit meerdere nodes. SensorScheme verbetert op van de stand van de techniek door het beschikbaar maken van technieken zoals *closures* en *continuations*, en programmeer-abstracties zoals hogere orde functies. Bovendien maakt SensorScheme gebruik van de eigenschap van *homoiconiciteit*, voor draadloos transport en dynamisch laden van complete programma's of slechts enkele functies.

- De beschikbaarheid van deze uitgebreide set van programmeer-abstracties en technieken maakt het mogelijk om bondig sensornetwerk toepassingen uit te drukken. Dit vermindert het geheugen en de communicatie-bandbreedte eisen voor de opslag en het transport van applicaties.

- SensorScheme introduceert ObjectStreams, een communicatie-techniek die *serialisatie* gebruikt voor het automatisch coderen en decoderen van berichten. ObjectStreams is geïntegreerd in de taal en de interpreter om het vervoer van programma's in het netwerk te vergemakkelijken. Het vereenvoudigt bovendien het schrijven van programma's door het verwijderen van de afhankelijkheid van de pakket-grootte van de onderliggende communicatie-hardware.

- SensorScheme introduceert een methode van macro-programmering van netwerken van heterogene sensor nodes. Het vereenvoudigt de bouw en het onderhoud van deze netwerken door het schrijven van een netwerk-breed programma dat de functionaliteit van alle apparaten in het netwerk bevat. *Specialisatie* van dit netwerk-brede programma (door *partiële evaluatie*) produceert node-specifieke programma's die alleen de functionaliteit bevat vereist voor ieder individuele apparaat, wat resulteert in aanzienlijke verkleining in vergelijking met het gehele netwerk-brede programma.

Wij motiveren en illustreren het ontwerp van SensorScheme door gebruik te maken van vier applicatie-scenario's, waarvan twee uitvoerig zijn behandeld in de literatuur, en de andere twee bevatten uitdagingen van de programmeerbaarheid ongeadresseerd in de stand van de techniek.

Met behulp van voorbeeld-implementaties van elk van de scenario's presenteert dit werk de programmeer-technieken en abstracties van SensorScheme, gericht op het verminderen van de complexiteit van het schrijven van sensornetwerk toepassingen.

We evalueren ook het platform met behulp van deze SensorScheme voorbeeld-implementaties. We tonen de geschiktheid ervan voor draadloze sensor netwerk platforms op het gebied van communicatie, energie, rekensnelheid en geheugengebruik voor de gepresenteerde toepassingen. Tevens beoordelen wij SensorScheme's gemak van programmering gemeten als de programma-groottes ten opzichte van de stand van de techniek.

Zoals meestal het geval is, moet een hoger niveau van abstractie worden afgewogen tegen een verhoging van het verbruik van rekentijd, geheugengebruik, communicatie-bandbreedte, en uiteindelijk energie.

Onze resultaten tonen aan dat de extra rekentijd veroorzaakt door de interpreter aanvaardbaar is voor typische WSN scenario's, waardoor er geen significante vertraging is en slechts een kleine toename van het energieverbruik.

Vervolgens laten we zien dat hoewel ObjectStreams communicatie vereist dat communicatie protocollen enigszins moeten worden geherstructureerd, het vergelijkbare communicatie-eisen voor een soortgelijke toepassing biedt.

Bovendien tonen we aan dat geïnterpreteerde SensorScheme programma's aanzienlijk korter kunnen zijn dan hun compileerde tegenhangers. Dit maakt het mogelijk om een breder scala van toepassingen en functionaliteit te bieden met het beschikbare geheugen van WSN-apparaten, en het makkelijker te maken om deze programma's te schrijven.

Concluderend, de voordelen voor de programmeerbaarheid van de dynamische, high-level talen zijn binnen bereik zelfs voor de kleine, simpele apparaten zoals gebruikt in draadloze sensor netwerken; zulke talen bieden niveaus van functionaliteit die niet op andere andere manieren beschikbaar zijn.

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

Recent technological advances in low power digital radio transmitters and receivers, microelectromechanical systems (MEMS) sensors and low power silicon integrated circuits bring closer the image of “calm technology that recedes into the background of our lives” [Wei96], as expressed by Mark Weiser, one of the first visionaries in the field of *Ubiquitous computing.* It has created a new domain of computing, *wireless sensor networks* (WSNs) that sense their environment, and collectively compute and reason upon the perceived state of the world around them. Already, WSNs have found applications in the field of environmental monitoring, and smart buildings. In the foreseeable future wireless sensor networks can also have a great impact in the supply chain management business. WSN nodes attached to crates, roll containers, pallets, and shipping containers can monitor the transportation process, and raise an alarm when the transport plan is not properly executed.

1.1 WSN research

Building WSN applications involves many of the hard topics in computer science, such as distributed and parallel computing, reliability and redundancy, network protocol design, and real time interaction. The hardware platforms used, with only minimal resources, have necessitated a full redesign and implementation of all basic functionality, especially scheduling and synchronization, and communication protocols, with strong attention for efficiency.

At the start of the work laid out in this dissertation the research in these fields had started to take shape, and early results were reported. A variety of hardware platforms for WSNs had been created, and operating systems, communication protocols and support software had been designed from the ground up to enable use of these hardware platforms. These early results demonstrated that the key to success is compactness and efficiency.

In the same time, however, these early successes raised the issue of the need of powerful high-level programming methods and abstractions, to reduce the complexity of the applications and bring programming these networks of computing within the reach of a wider audience.

Most importantly, a clear need emerged to program the devices after deployment, when the devices are accessible only through their wireless communication interface. While there is a clear need to (re)programming devices using the wireless connection, it also poses a number of risks and challenges.

First and foremost, facilitating wireless reprograming necessarily reduces the efficiency and increases size of programs running on these devices. The small size and low cost of WSN platforms necessitates efficiency and small size. Inclusion of reprogramming facilities forces design trade-offs to keep the devices usable and useful. Next, the possibility of wireless access to the devices while they are deployed – and therefore publicly accessible – raises a number of issues regarding security and protection, that should be taken into account.

Furthermore, one of the research areas that receives much attention is the question how to write programs for wireless sensor networks. While this is an ongoing debate in any field of computer science, for sensor networks the question is even more pressing, because wireless sensor networks combine a number of properties in a unique way:

Inherently distributed
: Each device is strongly coupled to a physical location or object. Therefore the WSN platform consists of a multitude of interconnected devices.

Data parallelism
: Especially in environmental monitoring applications, all nodes execute essentially the same program, each operating on different data, received through its sensors.

Unreliable, dynamic network
: In contrast to conventional distributed and data-parallel systems, the network connecting WSNs is not a dependable system interconnect, but a low-bandwidth, high-loss connection. Moreover, the structure of the net-

work – which nodes are connected to each other – may change continuously to some extent, even for non-mobile networks.

Autonomous operation
: The projected size of networks precludes manual configuration. The network must run unaided continuously for a long stretches of time, while coping with changing network connections and additions and removals of nodes.

End-user programmability
: Using a wireless sensor network essentially means programming it to perform a task autonomously without user interaction. Essentially this is a form of programming that has to be performed by the users of the network themselves.

While each of these issues appear to be hard problems by themselves, wireless sensor networks combine all of these together.

1.2 Focus of thesis

This dissertation proposes a new solution to the above–mentioned problems. The proposed solution is practical and application-oriented: it attempts to provide efficient solutions to a set of application scenarios given the availability of wireless sensor network hardware that makes small and abundant wirelessly communicating and computing devices both possible and economically feasible.

In this dissertation we focus specifically on sensor nodes like the Mica [HC02] and Telos [PSC05] motes. Larger device classes, such as the Intel Imote 2 [Croa] do not have the severe memory and processing limitations of the smaller motes, but we focus on these smaller platforms because, even as Moore's law progresses, they are more competitive in terms of price, energy use and ultimately cost of ownership. These platforms have the following limitations:

Limited energy budget
: Typical usage scenarios for WSNs demand a long battery lifetime, since frequent replacement or recharging would be too impractical or costly. All aspects of WSN software design are therefore focused towards minimizing energy use. Most energy is consumed by the radio hardware and sensors, so minimizing the use of those is the most effective way to minimize energy use.

Very limited memory
: The target platforms contain typically less than 64 KB of program memory, and 10 KB of RAM or less. These limits are hard, since no virtual memory mechanisms are available. Current practice shows that the limited RAM is most pressing, since WSN applications are usually simple and small enough to fit in program memory.

Simple, low speed CPUs (< 10 MHz 8/16 bits)
: This limitation is the least challenging one. Current sensor network applications typically have CPU usage of only a few percent, partly because the small working memory limits the data available for processing, and, more importantly, duty cycles are low as a strategy for minimizing energy use. Furthermore, CPU cycles are comparatively cheap. On current WSN platforms, sending a single message and receiving it on another node takes the amount of energy equivalent to as much as half a second of computation time.

1.2.1 Naming

The hardware platform on which this research is based, is referred to in the literature by a variety of names. *Mote* is the name used for some of the early platforms. More generally, *node* is used as an indication of the devices' role as part of a network, with derived uses like *sensor node* or simply *sensor*, as explicit references to the embedded sensors and the *wireless sensor network* of which they are part.

For certain application domains, typically more interactive and heterogeneous such as smart buildings, the collective set of such devices are referred to as *smart object systems* or *cyber-physical systems* and an individual device an *object* or *artifact.*

Still, the term *wireless sensor network* is more widely used and agreed upon. In this dissertation, we will refer to collections of devices as *wireless sensor networks* or *WSNs* for short. Individual devices will be referred to as *sensor node* or *node* in short, or simply as *device.*

1.2.2 SensorScheme

This dissertation presents a programming language and runtime environment called SensorScheme, especially designed to allow sensor network applications to be loaded onto the nodes in a deployed network using the wireless interface.

SensorScheme's design produces small programs that can be loaded onto the network in a variety of ways – either to individual nodes, several nodes together or onto the entire network. Together this ensures the process of programming or reprogramming sensor nodes to be efficient and adaptable to its use.

SensorScheme is designed to be easily programmable, resulting in small and easy to understand programs. We have achieved this by providing a level of abstraction matching WSN applications, and allowing programmers to add new abstractions themselves. SensorScheme incorporates the state of the art of programming abstractions for wireless sensor networks, and adds new ones to further simplify the task of writing a wireless sensor network program.

As is generally the case, the benefit of higher abstractions is traded for reduced efficiency of execution time and memory use. For the resource-lean platforms of wireless sensor networks, efficiency is of special importance. Therefore, during the design and implementation of SensorScheme efficiency has received much attention, and we will show that performance degradation is within acceptable limits.

1.3 Contributions

This dissertation proposes a complete software platform taking into account many of the research topics receiving much attention in the WSN community, and makes a number of contributions in some of these areas. Even though these topics are well-researched, this dissertation uses a different approach to abstractions for WSN applications, thereby creating solutions unique in the WSN architecture landscape.

The research contributions of this dissertation are the following:

1. The first contribution of SensorScheme is to enable loading and programming wireless sensor nodes using wireless communication. This has been achieved for severely restricted hardware platforms intended to be small, cheap and energy-efficient.

    (a) Wireless programming of the nodes is safe, without the risk of damage or dysfunction of the nodes caused by loading faulty or malicious programs.

    (b) Programs to be loaded wirelessly are platform independent, and may execute on devices with different hardware characteristics such as

CPU architecture, amount of memory, wireless communication peripherals, and the number and kind of sensors, actuators and other peripherals.

(c) Wirelessly loaded programs for typical WSN applications are small, which benefits the speed and energy efficiency of the loading process and preserves memory.

2. The second contribution concerns the methods of building wirelessly loadable programs. SensorScheme incorporates many of the techniques and abstractions present in the state of the art to reduce the complexity and size of wireless sensor network programs. While these have individually been available in other systems, SensorScheme is the first platform to provide these techniques together in a wirelessly reprogrammable system. Additionally, SensorScheme introduces new tools and abstractions to further improve the usability of wireless sensor networks.

   (a) SensorScheme enables use of the functional programming paradigm, which aids to minimize the size of programs, and facilitates the use of *reduction strategies* (see Section 2.5.2) in WSN programs.

   (b) SensorScheme offers program structuring techniques such as *closures* and *continuations* (explained in Chapter 5) that can be used to emulate multi-threading and object orientation.

   (c) SensorScheme includes ObjectStreams, a communication mechanism that allows programs to communicate messages containing language-level values rather than arrays of bytes. Messages may be of arbitrary size, possibly occupying multiple radio packets, without knowledge or concern to the programmer for filling and transmitting the individual packets. To our knowledge no other WSN operating systems, communication libraries and abstractions have aimed to provide similar convenience and platform independence.

   (d) SensorScheme is extended with a partial evaluator to specialize a general program describing the behavior of the entire network into node-specific variants that are significantly shorter. We use this specialization method as a new way to macro-program heterogeneous sensor networks.

1.4 Structure of thesis

Chapter 2 describes the background of the field of Wireless Sensor Networks research: The applications expected to be addressed using this technology; the embedded computer devices in use, their computational resources and energy consumption. It furthermore reviews the state of the art in WSN research of topics that are relevant for the material presented in this dissertation.

Chapter 3 describes some concrete application scenarios which we will use to illustrate the concepts and design trade-offs of the software architecture described in later chapters, and to derive concrete requirements for the described software architecture to be validated in Chapter 9.

Chapter 4 motivates the architectural decisions of our software platform, based on the requirements posed by the application scenario's of Chapter 3 and the existing solutions described in Chapter 2.

Chapter 5 introduces the use of the SensorScheme platform by presenting implementations for the application scenarios of Chapter 3.

Chapter 6 discusses the design of the SensorScheme program interpreter, memory organization and data types, and the overall system architecture.

Chapter 7 describes the ObjectStreams communication mechanism developed for the SensorScheme platform.

Chapter 8 describes an extension to the SensorScheme architecture of previous chapters to facilitate macro-programming of heterogeneous sensor networks.

Chapter 9 evaluates various aspects of the SensorScheme platform by measuring computation time, memory use, code size and communication overhead.

Chapter 10 concludes this work.

1.5 List of Publications

This dissertation is based on previously published technical reports or conference proceedings. The material presented in the subsequent chapters is based on the content of these publications.

- L. Evers, J. Kuper, Partially Evaluated Sensor Networks: Automatic Specialization for Heterogeneous Wireless Sensor & Actuator Networks. In: Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, january 19-20, 2009, Savannah, GA, USA. pp. 73-80. [EK09]

- L. Evers, M.E.M. Lijding, J. Kuper, Generic Multi–Packet Communication through Object Serialization. In: Proceedings of the 3rd international workshop on Middleware for sensor networks, 1-5 december 2008, Leuven, Belgium. pp. 25-30. [ELK08]
- L. Evers, P.J.M. Havinga, J. Kuper, Dynamic Sensor Network Reprogramming using SensorScheme. In: Proceedings of the 18th Annual IEEE Symposium on Personal, Indoor and Mobile Radio Communications, 3-7 september 2007, Athens, Greece. pp. 1-5. IEEE Computer Society Press. [EHK07a]
- L. Evers, P.J.M. Havinga, J. Kuper, Flexible Sensor Network Reprogramming for Logistics. Technical Report TR-CTIT-07-51, Centre for Telematics and Information Technology, University of Twente, Enschede. [EHK07b]
- L. Evers, P.J.M. Havinga, J. Kuper, Flexible Sensor Network Reprogramming for Logistics. In: Proceedings of the Fourth IEEE International Conference on Mobile Ad-hoc and Sensor Systems, MASS 2007, 8 - 11 October 2007, Pisa, Italy. IEEE Computer Society Press. [EHK07c]
- L. Evers, P.J.M. Havinga, J. Kuper, M.E.M. Lijding, N. Meratnia, SensorScheme: Supply Chain Management Automation using Wireless Sensor Networks. In: Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2007, 25-28 sept 2007, Patras, Greece. pp. 448-455. IEEE Computer Society Press. [EHK$^+$07d]
- L. Evers, M. J. J. Bijl, M. Marin-Perianu, R. Marin-Perianu, P. J. M. Havinga, Wireless Sensor Networks and Beyond: A Case Study on Transport and Logistics. International Workshop on Wireless Ad-hoc Networks (IWWAN 2005), London UK, May 23-26, 2005. [EBMP$^+$05]

Chapter 2

State of the Art

This chapter presents the state of the art in different areas of WSN research that bear relevance to the topics discussed in this dissertation. It starts with a description of the applications that are expected to be addressed with WSN technology. Subsequently, this chapter reviews the hardware that makes up the devices that we study, past, present and future. Next, it discusses the software platforms developed for these hardware platforms and many of the relevant issues in designing software platforms for wireless sensor networks.

2.1 Applications

Before we take a look at the technical aspects of wireless sensor networks, this section gives an account of the sort of applications researchers have attempted to realize. Römer and Mattern [RM04] present a survey on the large and varied application space for wireless sensor networks. Here we present a similar account, with the intent to give an indication of the types of problems researchers have tried to solve using wireless sensor network technology.

2.1.1 Battlefield monitoring and border patrol

Early research on sensor networks has been inspired by an application that has received much attention: Using small sensor devices to monitor a battlefield, country border or other large area and report to military personnel when persons or vehicles enter an area and keep track of their movements.

The proposed method to achieve this is by deploying wireless sensor nodes into the monitored area either by hand or dropped from the air. The nodes use a number of sensors such as magnetometers and micro-radars to detect vehicles. They communicate among each other to determine an accurate location of the vehicle, before transmitting this location to a human operator outside the network.

One of the first academic research projects – the *Smart Dust* project [WLLP01] at the University of California, Berkeley targeted this application scenario. It resulted in a small field experiment, using both hand-placed as well as air-dropped sensor nodes capable of detecting vehicles.

Later projects focused on this scenario, improving various aspects of hardware design [HKS+04, ARE+05], or on algorithms and protocols to improve target detection and tracking [The03, GJP+06, NW04, Röm04, LCL+03, GKGM05, JSS00, WM04, BHS03, KR05, SLO05, SHW08, BGL+07, YS03, GBCT06, LC02, WSBC04, LRZ03, ABC+04].

Chapter 3 presents this application as one of the example scenario's used throughout this thesis.

2.1.2 Environmental monitoring

One of the most prominent applications of wireless sensor networks is its use in environmental monitoring. To this end, devices periodically sense environmental parameters such as light, temperature and humidity, and transfer the sensed data to a computer outside the network, where it can be stored in a database or processed otherwise. Various methods have been investigated to transport only a summary or subset of the data, to reduce the traffic across the wireless network and extend battery life of the nodes.

One method of communication reduction stands out in particular, because of its broad acceptance and frequent use. Using this method the network is arranged in a tree-shaped structure, where every node selects a parent node that connects it to the network's root node, possibly via multiple transitions or *hops*. When it is acceptable to receive only summary information from the network, such as maximum, minimum or average values (or any other associative and commutative function), each intermediate node calculates the summary of data received from its child nodes. This method is generally referred to as *aggregation*.

A sizable number of test beds with deployed networks have been reported already, monitoring environments as diverse as bird nesting sites [MCPA02], vineyards [Int], redwood forests [Yan03], bridges [JDK+05], potato fields [LBV06], coral reefs [ZCH07] and volcano's [WAJR$^+$05]. We also use this application as one of the scenarios described in Chapter 3.

2.1.3 Forest fire detection

An environmental monitoring application that has caught special attention is the use of wireless sensor networks to detect forest fires. While this bears many similarities with other environmental applications, it deserves special attention due to the specific interest by many researchers [Fir, BHS03, KFG$^+$03, FRL05]. The application is also somewhat different in nature, in the sense that instead of regularly reporting the environmental parameters, the sensor network only needs to report the (very rare) event of a detected fire.

2.1.4 Mobile wildlife monitoring

Aside from the previously mentioned applications where the network is deployed on a fixed location, more mobile monitoring scenarios have received attention. One of the earliest was the *Zebranet* [JOW$^+$02] project, using sensor nodes attached to a herd of zebras to track their movement, in January of 2004. Later projects have used WSN technology to monitor cattle [May04].

2.1.5 Smart building applications

WSN technology has been considered valuable in other places as well, including well-accessable and connected places like homes and offices. Connor *et al.* [CHK$^+$04] present two office applications. The first one uses WSN technology to keep track of available meeting rooms and to inform workers using status nodes in the hallways of the building. The second application, *Follow-Me* is an active visitor guidance system to assist visitors in navigating through a building.

Smart homes and offices, containing wirelessly connected sensors (ie. infrared presence detectors, break beam sensors, buttons and switches) and actuators (power switch, locks) in the home, kitchen or office to. These sensors can then be instructed to monitor the behavior of occupants, automate routine tasks to increase convenience or safety [BK06, AYKC04, VMKP03, LFO$^+$07, KOA$^+$99]. Within the smart home application space special attention is given to elder care [Sta02] as well.

A related application is smart parking spaces assisting in finding a parking space [CSC06, WZL06, KGMG07].

While most of the networks involved in these applications are static, these applications differ from environmental monitoring in their focus on a combination between sensing and actuation. Sensor data from nodes in the network flows to other nodes instead of only to outside network operators.

Chapter 3 describes a possible smart building application as one of the example scenarios.

2.1.6 Logistics

The use of wireless sensor network devices holds promise for the transport and logistics business. Besides the passive Radio Frequency Identification (RFID) chips are gaining acceptance in some companies, active communication and computation devices such as active sensor nodes may prove valuable as well. WSN devices are proposed for a variety of logistics tasks, ranging from electronic seals [DBK+04], to autonomous enforcement of security guidelines [Kno04], or tracking and tracing of shipments [SFCB05, MP04].

This work is partly based on some contributions involving logistics applications [EHK+07e, EBMP+05], and uses a logistics scenario in Chapter 3.

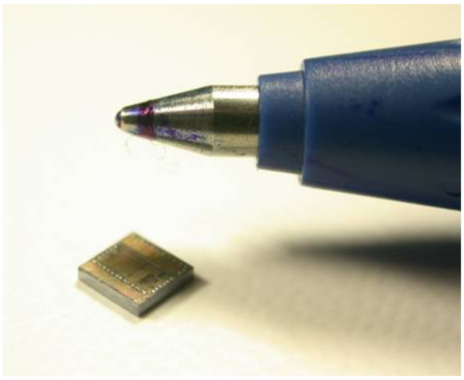
2.2 Hardware architecture

The vision of WSNs is to embed computing devices into the environment or inside daily objects. To make this practically feasible, the research focuses on use of small, battery-operated devices, accessible through wireless network connections. The devices may contain a variety of sensors to observe their environment, and are devoid of user interfaces like key pads or screens.

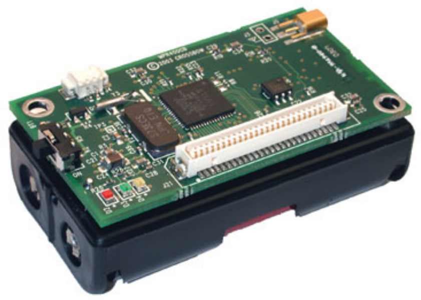
In contrast to other battery-powered devices like cell phones or PDAs, wireless sensor networks require long unattended operation. Frequent battery recharging or replacement is not possible once devices are deployed – especially in large geographic areas or in hard to reach places, such as in tree tops [Yan03] or out at sea [ZCH07]. Consequently, energy efficiency is of prime importance.

The devices must also be cheap, to allow their deployment in the quantities envisioned. The hardware platforms considered are therefore built out of some of the smallest and simplest components currently available.

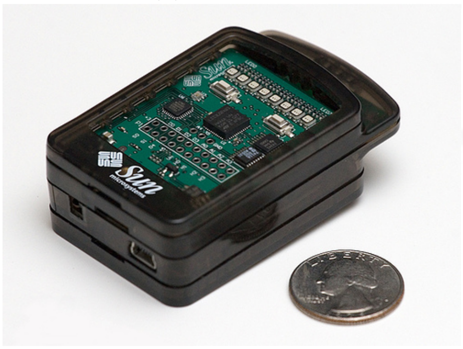
Figure 2.1 shows some of the devices that have been developed.



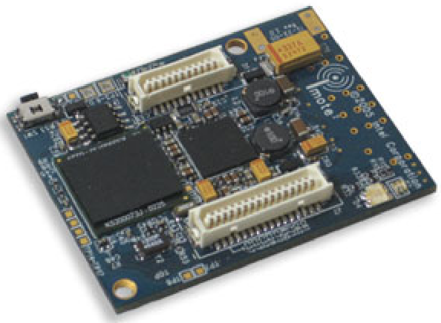
(a) Spec mote



(b) Mica2 mote



(c) Sun Spot



(d) Imote 2

Figure 2.1: Some of the devices developed for Wireless sensor network research

2.2.1 Platform classes

Hill *et al.* [HHKK04] review some of the platforms developed for wireless sensor network research, and group those into four different categories, based on their energy use and hardware characteristics. In this section we adopt their classification, but use different names, to be used for future reference in this dissertation. Table 2.1 lists a number of the developed devices, which we will use as examples.

Single chip sensor nodes. These are the first class of devices: custom single-chip IC's that merge computation, communication and sensing into a single chip. To date the *Spec* mote [Hil03] (see figure 2.1 a, table 2.1.1) serves as the primary example. It has very limited computational resources (CPU and RAM) and low power radio, and communicates only to more powerful devices. This class of devices is custom-built to achieve minimal form factor and energy use, at the expense of very limited computational resources, and may be used to address only the simplest of tasks. Currently these devices are unsuitable to create self-sustaining multi-hop networks, and may be usable only at the edges of a network.

Low power sensor nodes. This second class of devices are so-called COTS devices, built from **C**ommercial **O**ff-**T**he-**S**helf components. With energy use and price in mind, these devices are equipped with slow 8- or 16-bit microprocessors and a few kilobytes of working memory (RAM) and code memory (flash ROM).

The devices use low-power radio chips, capable of short-range, low data rate communication. Low power single chip radios are a relatively new development, and as such, standardization is just emerging. The prominent standard, IEEE 802.15.4 [IEE06], has been adopted by the WSN research community, and is now found in a number of different low power sensor nodes.

These nodes usually have a form factor that matches the battery size intended to power them. Input and output capabilities are usually limited to a few LEDs to report software status and a variety of sensors, possibly on an external, detachable board. Beigl *et al.* [BKZD04] provide a comprehensive account of the various sensors in use for various devices and applications.

Made from off-the-shelf components that are simple and cheap, these devices are a cost-effective platform that is still small enough for many applications, but is able to carry somewhat more energy and provides enough computational

Mote	CPU	Memory	Radio	Power[a]	Lifetime[b]	Price[c]
Single chip sensor nodes						
1. Spec [Hil03] 2003	8-bit Custom ISA 4-8 MHz	3 KB RAM	On-chip 916 MHz 50-100 kbps	3 mA act. 3 uA idle	1933 h	
Low power sensor nodes						
2. WeC [Hol00] 1998	8-bit Atmel AVR 4 MHz	0.5 KB RAM 8 KB Flash 32 KB ext. Flash	TR1000 916 MHz 10 kbps	18.8 mA act. 45.7 uA idle	131 h	
3. Mica [HC02] 2001	8-bit Atmel AVR 4 MHz	4 KB RAM 128 KB Flash 512 KB ext. Flash	TR1000 916 MHz 40 kbps	18.8 mA act. 35.7 uA idle	166 h	$115.00
4. Mica2 [Crob] 2002	8-bit Atmel AVR 7.37 MHz	4 KB RAM 128 KB Flash 512 KB ext. Flash	CC1000 916 MHz 38.4 kbps	26.63 mA act. 15.86 uA idle	354 h	$115.00
5. MicaZ [Tec] 2004	8-bit Atmel AVR 7.37 MHz	8 KB RAM 128 KB Flash 512 KB ext. Flash	CC2420 802.15.4 250 kbps	29.83 mA act. 16 uA idle	347 h	$99.00
6. Tmote [Mot06] 2004	16-bit TI MSP430 8 MHz	10 KB RAM 48 KB Flash 1 MB ext. Flash	CC2420 802.15.4 250 kbps	21.8 mA act. 5.1 uA idle	977 h	$78.00
High speed sensor nodes						
7. Btnode v3 [BKM+04] 2003	8-bit Atmel AVR 7.37 MHz	4 KB RAM 128 KB Flash 180 KB ext. SRAM	Bluetooth / TR 1000 38.4 kbps	41 mA act. 151 uA idle	40 h	$230.00
8. XYZ [LS05] 2005	32-bit ARM Thumb 1-58 MHz	32 KB RAM 256 KB FLASH 256 KB ext. SRAM	CC2420 802.15.4 250 kbps	72.28 mA act. 30 uA idle	181 h	$150.00
9. Sun Spot [Pro09] 2006	32-bit ARM Thumb 180 MHz	512 KB RAM 4 MB Flash	CC2420 802.15.4 250 kbps	98 mA act. 33 uA idle	160 h	$250.00
10. Imote 2 [Croa] 2006	32-bit ARM XScale 13-416 MHz	256 KB RAM 32 MB Flash 32 MB ext. SDRAM	CC2420 802.15.4 250 kbps	66 mA act. 390 uA idle	16 h	$299.00

[a] Values obtained from device specification if available or data sheets of main components. Active power taken when radio listening and CPU active.

[b] Calculated approximate life time when powered at 3.3V by 2 AA batteries of 2900 mAh each, with duty cycle of 5 % active and 95 % idle power.

[c] Commercial price as of October 2009 or price at latest selling date.

Table 2.1: WSN devices

resources to create ad-hoc networks and carry out computations on the sensed data.

High speed sensor nodes. A third class of devices has been developed, with faster 32-bit processors and more memory. These devices are also significantly more expensive and consume more energy. Early devices have been designed with higher data rate radios, such as Bluetooth [Bha01] (for example Btnode v3, see table 2.1.7). The availability of low power IEEE 802.15.4-compatible radio chips has reduced this need.

For tasks that require more computational resources, devices like these might be an alternative, at the expense of increased energy use. Table 2.1 also gives a rough indication of the lifetime of all devices. As an example, the lifetime of the *Tmote* low power sensor node is roughly 6 times longer than the *Sun Spot* (see figure 2.1 c, table 2.1.9) high power sensor node. To achieve similar life time, a battery pack six times as large may be used, which significantly increases form factor and price.

These devices provide a good alternative where cost and form factor are not the crucial limitations, and more computational power is needed than low power nodes can provide.

Other devices. Besides the device classes mentioned here other devices have been used and developed for WSN research. First, so-called gateway devices have been developed that connect a sensor network to the outside world. These devices are equipped with high speed wired or wireless network connections, are usually not battery-powered, and have computational resources ranging from PDA's [Croc] to desktop or server PC's.

Next, PDA's, mobile phones or other commercially available devices have been used in some research projects [BHS03, ZSLM04] as prototyping platforms. Results obtained using these devices might not readily translate to actual deployments using low power sensor nodes, however.

2.3 Future developments

Developments in electronics hardware are continual, as history has shown us. Software in development today will be commercially deployed on the next generation of computing hardware. It is therefore essential to keep in mind the characteristics of the next generation of WSN platforms.

For more than forty years advances in computing hardware has followed an exponential growth. In 1965, Gordon Moore expressed the observation that

> "the complexity for minimum component costs has increased at a rate of roughly a factor of two per year"

Historical data has shown this trend to be accurate for the last forty years. The continuation of this trend (at least for the coming 5-10 years[1]) yields ever faster devices with more memory for the same price. For WSNs this trend unfolds in two different directions: Future generations of hardware platforms will be more powerful than current ones at the same price level. Essentially, for the three device classes described before, we may expect each class to be equipped with the CPU speed and memory sizes of the next 'higher' device class.

Alternatively, the effects of Moore's law will fraction the price of WSN platforms for the same performance. This makes wireless sensor networks affordable in ever greater quantities, which may broaden the applicability of the WSN's into ever more areas.

2.3.1 Energy use

The dependency on battery-supplied power makes energy efficiency a major requirement for WSN platforms. The continuation of Moore's law also has consequences for energy consumption of integrated circuits. The advances in transistor count per chip are realized through miniaturization: subsequent generations of integrated circuits contain transistors of ever decreasing feature size. Where the energy use of the subsequent processor generations has continued to increase, the energy used per transistor has dramatically decreased as a consequence of miniaturization.

Besides reducing per-gate energy use, focus on energy-efficiency in the design of these chips can have great impact on the total energy used. A variety of low-power idle modes, a reduction of power mode transition delays, and powering down individual on-chip peripherals can greatly reduce real energy use, by keeping the device in a low power state for a greater fraction of time.

Again, for the future of WSN platforms the consequences are twofold: on the one hand devices of same speed and memory size will become more energy-efficient. On the other hand, for the same energy budget it is feasible to use

[1]Many predictions have been made when this exponential growth trend will halt, due to the physical limits of the shrinking feature sizes. It is unclear though whether exponential growth will cease soon, or new technological advances will keep enable continued growth of the number of transistors per chip.

higher performance CPUs. Already the trend of reducing energy use of subsequent generations of WSN platforms is apparent in the devices shown in table 2.1. In the six years between the WeC (see table 2.1.2) and Tmote (see table 2.1.6) platforms, energy use has decreased roughly sevenfold (when comparing only theoretical energy use calculated from the devices' data sheets); furthermore, the more powerful and capacious Sun Spots (table 2.1.9) are less power-hungry already than the WeC mote of eight years earlier.

Further advances in integrated circuit design and focus on low power has already decreased total power consumption significantly. Newer devices have lower idle power consumption (such as the TMote platform, table 2.1.6). This may reduce even further, since idle power consumption is mostly the result of current leakage in memory cells. Qin *et al.* show that with reduced operating voltages SRAM cell leakage can be reduced up to 90 %.

Future developments will make it possible to decrease active mode power consumption as well, with as much as three orders of magnitude for low-speed processors, as Nazhandali *et al.* show [NMZ$^+$05].

2.3.2 Radio

While WSN CPUs have been getting more energy-efficient, and can be expected to do so in the future, low power radio chips do not follow this trend. The wireless transmission of bits over a certain short distance is associated with intrinsic costs. The transmitted signal has to be powerful enough at the receiver to be discernible from background noise. The power consumed while listening for incoming packets may still reduce, however [WSGLA08].

2.3.3 Batteries

The power source used for WSN nodes are batteries. Their capacity, size, weight and cost are important factors in the usability of wireless sensor network technology. Powers [Pow95] presents an outlook into the future developments of battery technology. History has shown an increase in energy density of batteries, partly due to the use of new materials, such as lithium. Progress is, however, much slower than the rate of progress for electronics circuits, at around 5 % to 10 % increase per year.

2.3.4 Outlook

Currently developed WSN devices are designed with energy-efficiency as primary concern. In the future, we may expect such devices to be even more efficient.

The hardware platform of choice in the future may be similar in computational resources to the current low power sensor nodes for large-scale deployments, that require minimal cost and energy consumption. Recently, single chip solutions combining a CPU with radio have become available already, delivering the computational resources of low power sensor nodes in a form factor of single chip sensor nodes. For applications that can accept somewhat higher prices, larger 32-bit devices will provide more computational resources at significantly lower energy levels than current devices.

The target platforms to consider for the future have resources similar to the current low power sensor nodes as well as high speed sensor nodes. We will consider both of these device classes as the target for our software platform described in Chapter 6. This means that such a software platform must be designed to cope with the minimal computational resources of Low Power sensor nodes, while capable of making use of the less limited resources of high speed nodes if available.

While the energy cost per instruction for computation tasks will reduce dramatically, radio-communication involves inherent energy consumption. As a result, most of the nodes' energy will be spent on communication, while computation will be comparatively cheap. For software designers the task of developing an energy-efficient system will mean primarily to minimize the use of communication, while a reduction of computation time will have only little effect on energy use. We will take these observations into account for the design of our wireless sensor network platform in this dissertation.

2.4 WSN Operating Systems

The hardware platforms we discussed in the previous section require specialized software to bring about their intended behavior. The memory size and speed of some of the high speed sensor nodes allows the use of embedded systems variants of main-stream operating systems, specifically Linux.

The low power sensor nodes, which have our focus, require custom operating systems that minimize energy consumption and keep memory use and code size to a minimum, and incorporate design decisions that reflect the central role of communication in wireless sensor networks. A sizable number of sensor network-specific operating systems have emerged as research effort, starting with *TinyOS* [HSW+00], which is perhaps the most widely used. Others include *Contiki* [DGV04], *SOS* [HKS+05], *Mantis* [BCD+05] and *BTNut* [btn09].

The extremely limited hardware capabilities influenced the design of these platforms to a great extent. What is commonly called an *operating system* for wireless sensor networks and embedded systems in general is essentially a task scheduling routine and a library of source files containing device drivers to control peripheral devices.

2.5 Communication

As the sole method for performing data input and output, wireless communication plays a crucially important role in wireless sensor networks. This is reflected both in the communication-centric design of WSN operating systems and platforms and the large body of research on communication protocols for WSNs and related topics.

WSN platforms communicate at low data rates – at most 250 kbps raw data rate as table 2.1 shows, and over short distances of at most 30 – 100 meters. For networks that cover large geographic areas, nodes in the network can communicate to others only when intermediate nodes receive and forward the messages, possibly across multiple hops. In many static networks nodes build and maintain a *routing tree* that is rooted at one or more special *gateway* nodes that are connected to a base station – a larger computer, usually a regular desktop or laptop PC, from which the network is controlled and that receives the data produced by the network.

Typical WSN deployments have a dense network structure, where a node can receive from multiple other nodes in its vicinity – its *neighbors*.

The properties of the wireless medium cause wireless communication to be quite unreliable. While neighbor nodes may be able to communicate with each other some of the time, at times packets between neighbors may not be received without error. Nodes at somewhat greater distances, with which communication is not possible may, however, cause interference on a node that is listening to another node that is transmitting at the same time.

The individual network packets are very small – TinyOS uses a default message size of only 48 bytes; the maximum size supported by the IEEE 802.15.4 standard is 127 bytes. Short packets increase the likeliness that an entire packet will be received error-free. Short packets also reduce the time a node is transmitting to a minimum, which conserves energy, and leaves the shared communication medium free for other nodes.

The effective transfer rate of an individual sensor node across the network up to the base station is far lower than the raw data rate, and may be as little as a few bytes per second. The reason is that the communication medium is shared between many nodes, and only one can transmit at a time to avoid interference at the receivers. Using a multi-hop routing tree, messages need to be transmitted multiple times from hop to hop all the way to the base station. This low data throughput rate has fueled WSN research into devising protocols that are efficient, communicating as little as possible. In many cases, only a subset or summary of the sensed data is required at the base station. Sending only the required data greatly reduce the data communicated in the network.

2.5.1 Protocols

As is common in network communication, WSN communication protocols are organized as a stack consisting of a number of layers, each depending on the layers below it, and providing a communication service to the layer above it.

MAC Protocols. The lowest protocol layer, the *Medium Access Control* (MAC) protocol has been intensively researched and many alternatives proposed, specifically for sensor networks. Langendoen has presented a survey of the many MAC protocols for wireless sensor networks [Lan07]. The MAC protocol is important for WSN's because it has a large influence on the energy efficiency of the network. While nodes are not transmitting or receiving a packet, it is imperative that nodes put their radio into a low power mode to conserve energy.

Protocol layers above the MAC protocol serve a number of purposes, from simple broadcast communication among neighbor nodes to protocols transporting data to the base station, or in the reverse direction, from the base station into the network.

Protocol Stacks. The use of different stack layers is made very explicit in the design of the *Rime* [Dun07] protocol stack for the Contiki [DGV04] WSN operating system. The Rime protocols implement various kinds of protocols such as single-hop reliable unicast and multi-hop reliable bulk data transfer. The different stack layers are individually selectable for use by the application developer, and make use of each other to provide their communication services.

For the TinyOS WSN operating system, a similar set of communication protocols exists, as Madden *et al.* describe [LMG+04]. The authors discuss different communication services, such as single hop communication, various multi-hop communication services such as tree-based routing, intra-network routing and broadcast and epidemic protocols.

Routing Protocols. A number of distinct classes of network protocols has been proposed in research. *Routing* protocols form one of these classes. Akkaya and Younis [AY05] review and classify many of them. Routing protocols make sure that network packets destined for nodes that may be reachable only through several intermediate hops reach their destination. Intermediate nodes may decide where to forward the message using one of several methods: using a continuously maintained routing tree, by broadcasting the network to query for a path between source and destination, or using location information of the nodes in the network.

One routing protocol is of particular importance to this work, as we will make use of them and evaluate them in later chapters. This protocol is the *Mint* (from *Min*imum *t*ransmission) protocol distributed with the TinyOS 1.x source code [HSW$^+$00], and its successor *Collection Tree Protocol* or CTP for TinyOS 2.x. Mohan *et al.* evaluate and compare the Mint protocol extensively [MnWHG$^+$05]. The aim of these protocols is to construct a routing tree rooted at a node chosen by the application. To achieve this, every node in the network chooses a parent node from its neighbor nodes. The choice is based on the reliability of the communication link to the chosen neighbor such a way that data transmission to the root via the chosen neighbor will result in the shortest and most reliable end-to-end path. All nodes maintain a list of their neighbors and perform regular link quality measurements on which they base their choice for parent.

Transport Protocols Another important class of WSN communication are *transport* protocols. Wang *et al.* present a survey of a number of such protocols [WDLS06]. Transport protocols ensure reliable delivery of messages, and aim to minimize congestion in the network.

One of the transmission protocols reviewed by Wang *et al.* is the *Trickle* protocol. Trickle tries to update every node in a network with the latest version of some piece of data, for example some piece of configuration data or variable value. In case an update is made to the data, one of the nodes receives the update. Each node periodically advertises its data version, and if a neighbor possesses a newer version, it is requested to transmit it to all neighbors still holding expired versions. In turn, these nodes advertise their newly obtained data version, and will further propagate the new code. Given some time the whole network will be updated with the latest version of the data, without explicitly transmitting the code to any of them.

2.5.2 Communication abstractions

The communication protocols described above are a means of transferring individual packets – fixed-sized sequences of bytes – between a sender and receiver. Many sensor network programs require a different mode of communication, for which a number of *communication abstractions* have been proposed. This includes a number of works concerned with neighborhood interactions such as *Hood* [WSBC04], *abstract regions* [WM04] and *logical neighborhoods* [MP06]. These works make use of the observation that the decentralized computations performed by sensor nodes involve data from each node's *neighborhood*: a collection of other nodes that are reachable through direct communication or through a small number of hops. All nodes in the network make available some data – usually obtained from sensors – to the nodes in its neighborhood.

Communication may take place in the form of shared memory primitives, where nodes read and write shared variables, or by regular updates that are disseminated to all neighborhood members at constant intervals.

These neighborhood abstractions use a higher level of abstraction where they communicate not packets containing byte sequences of unknown content (to the communication layer), Instead they communicate values of a particular data type known at compile time, usually originating from sensors.

Typically, these neighborhood abstractions include functionality to efficiently perform calculations on these sensor values using explicit iteration over all neighbors' values, or implicitly though the use of the *reduction strategy* which calls a *reduction function* on all of the neighbors' values to obtain the sum, mean, or min/maximum.

In these systems the shared values are either of scalar type or flat structure types only, so as to fit inside a single packet. Exchange of multiple values or larger data sets is done through multiple read or write operations only, at the expense of communicating multiple packets.

2.6 Programming models

Wireless sensor networks are a new computing platform and as such receives much research attention trying to investigate what are appropriate methods to design and program this new platform.

The contributions of this are in various topics related to programming models and methods for sensor networks. In this section we mention the main topics investigated by the research community and point to some of the work in these areas. Subsequent chapters will discuss the topics mentioned here more thoroughly.

2.6.1 Multi-threading

The little available memory in low power sensor nodes makes supporting multi-threaded applications somewhat problematic, due to the memory required for threads and their runtime stacks. The availability of multi-threading can ease application development, however.

The *BTnut* [btn09] and *MANTIS* [BCD$^+$05] operating systems both support multithreading. *TinyThread* [MS06] is a thread library for TinyOS. With these systems, each thread requires its own stack. *Y-threads* [NPR06] and *Fibers* [WM04] are a more limited and memory-efficient method. Y-threads uses separate stacks for pre-emptable tasks, but uses a shared stack for *Run-to-Completion Routines* invoked from the tasks. Fibers allows for a single blocking I/O context only. Switching out of the blocking call requires only saving the register set to memory, and restoring it when switching back. *ProtoThreads* [DSVA06], part of the Contiki [DGV04] OS is the simplest solution to adding thread support. ProtoThreads aims to remove the split-phase nature of an application's main loop by using a combination of C macro's.

This work makes contributions in the area of multi-threading and concurrency, and a more in-depth analysis is presented in Chapter 5.

2.6.2 Wireless reprogramming

Most WSN operating systems (including TinyOS, Contiki, BTNut, and MANTIS) are programmed with a statically compiled code image. Nevertheless, a number of research projects have attempted to enable loading of applications or modules. Reprogramming the devices using the wireless link requires cooperation from the operating system to receive the new program, install it on the node and execute it. Additionally, the new program needs to be transmitted to the node using a suitable communication protocol.

The different methods of wireless reprogramming described in the literature concern the executable format of the new program. Chapter 4 extensively discusses their exact differences and benefits. In this section we will just mention a number of works that target wireless reprogramming for WSN platforms.

One of the first reprogramming works is XNP [Cro03] for TinyOS. It is able to wirelessly reprogram nodes that are in direct wireless connection with a base station. Deluge [HC04] and MNP [KW05] are later improvements that transport program images over multiple hops to the entire network.

Some works have aimed to reduce communication while transmitting new program images by updating the current program with the differences compared to the new version. A number of different methods have been published, similar to the Unix 'diff' utility [RL03], based on the *RSYNC* protocol [JC04], or other differential methods, such as MOAP [SHE03] and FlexCup [MGL+06].

Some WSN operating systems support loading of modules into a static kernel, which facilitates loading and updating applications on the devices. This goal was the primary motivator for the development of SOS [HKS+05]. Similarly, MANTIS [BCD+05] allows loading of program modules. Contiki [DFEV06] supports run-time loading of standard ELF files. The differential transmission method FlexCup [MGL+06] transports modules for the TinyOS-based system tinyCubus [MLM+05].

Virtual machines and interpreters for WSN platforms facilitate wirelessly loading programs onto nodes as well. Maté is the first virtual machine developed for wireless sensor networks. *Agilla* [FRL05] is a modification of the Maté VM that uses mobile agents as a means of program transport and communication. Similarly, *ActorNet* [KSMA06] is an agent-based platform for WSNs that uses the Scheme programming language to program agents. SensorWare [BHS03] is similar mobile agent-based system. Its mobile agents are programmed in the programming language TCL. Sun provides the SPOTs [Pro09] WSN platform that is programmed with the Java language, and contains a Java VM [LY99]. The JavaVM's high memory and computational resource requirements are reflected in the design of the Sun SPOT platform (see table 2.1.9). More recently, however, the Perk [Cor08] and Darjeeling [BLC09] virtual machines have managed to put a JavaVM onto low power sensor nodes. Contiki contains the *SCript* scripting language that can be used to load and interpret small scripts on nodes.

2.6.3 Macro-programming

One research direction taken aims to allow an application designer to program the network as a whole, instead of the individual devices separately. Many variations of this *macro-programming* model have been developed.

Macro-programming platforms typically use a high-level programming language to specify in-network computations. Communication is not explicit, but happens 'under the hood' as part of the execution of language-specific constructs. As an example, the *Regiment* [NW04] platform uses a lazy functional programming paradigm derived from the Haskell programming language, and incorporate the *abstract regions* neighborhood communication library, using *regions* as a fist class language element. Operations on *regions* cause communication between nodes in a region.

Kairos [GKGM05] and its successor *Pleiades* [KGMG07] add some abstractions to imperative base languages *Python* resp. *C*: reading and writing variables at nodes (using a `variable@node` syntax) and iterating through the one-hop neighbors of a node (using a `cfor` language construct), and automatic program partitioning and migration for minimizing energy consumption.

The use of a non-imperative high-level language with language support for sets of uniform data and streaming execution has been proposed in a variety of ways: Flask [MMWN07], *SOSNA* [KC08] and – as mentioned already – Regiment [NW04, NMW07], uses lazy streams inspired by the *Haskell* programming languages to perform computations on streams of sensor values.

Logic-based declarative languages, derived from *Prolog* have been proposed in several instances as well for *Cooperative Artefacts* [SGKK04, SKGK04], and as part of *Semantic Streams* [WZL06].

Besides writing high-level programs for *homogeneous* networks, where every node has the same task and receives the same program, macro-programming has also been applied to *heterogeneous* networks, where every node is associated with a specific task. RuleCaster [BK06], *snBench* and *Abstract Task Graph* [BPRL05] all use a declarative language to define the behavior of the network, which is then compiled into separate programs for each individual node. Again, communication is implicit in these macro-programs. The role of the compilers for these systems is to split the network-wide behavior into node-specific programs in such a way that total communication is minimized.

Chapter 8 presents our contribution in the area of macro-programming.

2.6.4 Distributed database view

Finally, a class of sensor network platforms uses a programming model where the sensor network is viewed as a distributed database of sensor values. This model of the network is strongly coupled with the environmental monitoring class of applications described in Section 2.1.2. in this model, the network produces a regular stream of sensor values that are *aggregated* with some operation, and forwarded to the base station through a routing tree as described in Section

2.5.1. A wide variety of implementations of this programming model exists, such as Cougar [BGS01], TinyDB [MFHW03] and SwissQM [MAK07].

We will use the distributed database view programming model in one of the application scenarios in Chapter 3.

2.7 Outlook

This chapter reviewed the research performed in the field of wireless sensor networks. First we have described WSN hardware platforms and current and potential applications that make use of these platforms. Subsequently we reviewed the contributions on software platforms, systems and languages for WSNs. While each of these works contribute to advance the state of the art on wireless sensor networks, due to software and hardware incompatibilities and resource constraints these individual works cannot be combined into a comprehensive software platform capable of addressing a multitude of WSN applications.

This dissertation describes the design of a WSN platform designed to host a wide range of WSN applications, taking into account the tools and techniques described in this chapter and the resource restrictions of current WSN hardware platforms. We use a set of representative application scenario's (Chapter 3) from which we derive design requirements (Chapter 4) for the SensorScheme platform described in Chapters 5 – 8. Finally we use the application scenarios to evaluate the performance and effectiveness of the SensorScheme platform in Chapter 9.

Chapter 3

Scenarios

This chapter defines four concrete wireless sensor network application scenarios. We will use these scenarios to define the requirements for the SensorScheme software platform (described in Chapter 4), and evaluate our platforms with the use of these scenarios (in Chapter 9).

Next, this chapter analyzes the common requirements for implementing these applications on a WSN platform and discusses design alternatives that may be used to implement these requirements. Using the conclusions from this analysis, Chapter 4 will further discuss the design tradeoffs made to create a WSN platform to run all of the application scenarios described in this chapter.

The set of applications we consider is diverse: from simple applications used merely as proof of concept to complex dynamic applications using a heterogeneous collection of devices. These applications, however, each represent a larger class of WSN applications, some of which have been identified in Section 2.1, and with a wide variety of requirements to the WSN software platform. We intend for these example application scenario's to be a representative set of applications for the entire WSN application spectrum and as such present a broad range of WSN platform design issues, and guide us towards solutions.

3.1 Intruder detection

One of the most cited applications for wireless sensor networks is battlefield monitoring, also known as *intruder detection* or the *pursuer evader game* (PEG). Section 2.1.1 already mentioned it. This application is used in a large body of research as a proof-of-concept example. We also include this application here for the same reason.

For the intruder detection application, a sensor network is spread out in a large outdoor area, that needs to be watched for intruding vehicles. All the nodes know their own location as a set of x- and y-coordinates. The sensor nodes are equipped with sensors that can sense the nearby presence of vehicles. We assume the use of a magnetometer, that produces sensor readings that are proportional to the distance from a sensed vehicle.

All sensor nodes periodically read out their magnetometer. If the reading is greater than a certain noise threshold, we assume a vehicle has been detected. All nodes in the vicinity of the vehicle will then transfer their sensor readings and location coordinates to each other, and one of them can calculate the estimated position of the vehicle as the *centroid* of all the measurements, using the magnetometer readings as weights.

Other work proposing implementations for this application use only each node's direct neighbors to gather sensor readings from. In our scenario nodes gather sensor data from their two–hop neighborhood (that is, a node's neighbors and its neighbors' neighbors).

For the transfer of sensor readings within the network, we use a gossip protocol to periodically have nodes broadcast their sensor data and location, and receive data from their neighbors. In every subsequent period, nodes broadcast all data received from its neighbors alongside their own new sensor data, and receive all second–hop–neighbor data sent by neighbors. Every period, one of the nodes calculates the vehicle's location using all first and second hop data received. The calculated location is then sent to a base station outside the network, from where it can be used to visually locate and approach the vehicle.

3.2 Environmental monitoring

Another application of wireless sensor networking technology is to monitor environmental conditions of the deployed area of the sensor network. In some situations the sensor data is not needed in raw, unprocessed form, but only summary information calculated from the readings. These calculations can take place on the nodes in the network as the data is transported to the base station. Calculating summary information in the network reduces the total data transmitted, making the network more energy-efficient and scalable.

Previous research has adopted a view of the sensor network as analogous to a real-time database and request data from it by way of *queries* – programmatic expressions of the operations to perform on the data as it travels through the network. This application scenario displays the capabilities of a WSN platform to dynamically change or extend the application's functionality through the use of a query language as a restricted (ie. not Turing-complete) programming language. The restrictions posed upon the query language are motivated by the application domain as well as the hardware and software restrictions o the WSN platform in use.

Subsequent implementations of this scenario in the state of the art show a gradual progression towards increased expressiveness of the query language, ranging from Cougar [BGS01] to TAG [MFHW02] and TinyDB [MFHW03], and used as an example application to demonstrate the use of *application specific virtual machines* in the form of QueryVM [LGC04] and the SwissQM [MAK07] virtual machine.

In analogy to the database query language SQL, the tinyDB query platform uses queries written in a special query language, *tinySQL*. Other systems use different query languages of similar nature. Queries can be dynamically loaded into the sensor network devices, and in most implementations, the network can process multiple queries simultaneously.

WSN querying applications consist of a number of parts: All nodes in the network sense the environment using their sensors. The sensor data is then sent to a data sink across a previously set up and continuously maintained routing tree. At each intermediate branch in the tree, a summary is calculated from the data from higher up in the tree. The summary data is then forwarded further on to the tree, all the way until the root at the base station.

We use this database abstraction as an application scenario to monitor the soil moisture in a crop field. We use an (imaginary) rectangular field of 2 hectares in size that contains 800 sensor nodes in a 5×5 m grid formation. We monitor the field with course-grain summary values of each of the eight 50×50 m sub-areas.

3.2.1 Overview query

Every 5 minutes we request the minimum, maximum and average soil moisture values of each of the sub-areas, which are stored into a database. We call this the *overview query*. As a tinySQL query this can be expressed as

```
SELECT min(moist), max(moist), avg(moist) FROM sensors
  GROUP BY  x / 10 + y / 10 * 4
  SAMPLE PERIOD 5m
```

We assume here that the x and y coordinates are given in meters with (0, 0) in one of the corners of the field and axes parallel to the field sides, and `/` is the integer division operator.

While stated using tinySQL syntax, the overview query does not conform to the TinyDB query requirements due to the complexity of the `GROUP BY` clause. In fact none of the platforms in the state of the art mentioned implementations allow for queries of this complexity. Chapter 5 shows how SensorScheme is able to execute this query, however.

3.2.2 Anomaly query

When anomalies are detected in the reported values of any of the eight sub-areas, a new query is sent into the network that senses the 100 individual sensor nodes in the anomalous sub-area for two hours. We can express this *anomaly query* as the following tinySQL query:

```
SELECT nodeid, moist FROM sensors
  WHERE x / 10 + y / 10 * 4 = 6
  SAMPLE PERIOD 30s
  DURATION 2h
```

We can group the computations of both the overview and anomaly queries into two parts: 1) generate data, by reading from sensors or other data sources (such as the constant coordinates), but only when the conditions in the **`WHERE`** clause are met; 2) aggregate received data, producing intermediate averages and grouping them by values produced in the **`GROUP BY`** clause. Additionally, the kind of aggregates that can be calculated are those for which the order of calculation and location (on which node does it take place) is irrelevant. Aggregations should therefore be limited to pure (without side-effects) commutative functions.

3.3 Tracing and monitoring in logistics

Besides the well-studied WSN applications for environmental monitoring, wireless sensor networks have the potential to make a great impact in the supply chain management business. WSN nodes can be attached to crates, roll containers, pallets, and shipping containers to actively track the transportation process. During transport the devices verify proper handling conditions of goods like temperature for fresh foods, and can detect correct placement in trucks or containers, and raise an alert otherwise. Actively monitoring every transported

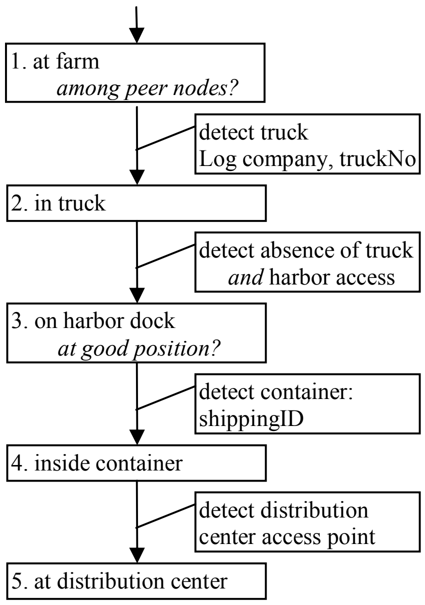


Figure 3.1: State diagram of the transportation process

item in this way can significantly reduce delivery delays and loss or theft of goods, which cause a significant loss of revenue.

This application scenario was first used in previous publications ([EHK07a], [EHK07c], [EHK$^+$07d]), on which this dissertation is based. In this work we have extended the application scenario and provide a full implementation in Chapter 5.

In this application scenario we track a shipment of bananas as it travels from the farm near Rio de Janeiro to a supermarket distribution center in Rotterdam. The bananas are packed in boxes stacked onto pallets, each equipped with a WSN node tracking its every move. From the farm, these pallets travel in trucks to a loading dock at the harbor, where they are loaded into shipping containers that carry them all the way to the supermarket chain's distribution center. During the whole trip, the bananas need to be kept cool, between 10 and 15 degrees Celsius, and away from sources of ethylene gas, such as fresh coffee beans, that adversely influence the ripening process.

Figure 3.1 shows a state diagram of the stages and transitions that these pallets will go through during the transportation process from the farm to the distribution center, which we'll call a *journey.*

While a pallet is waiting at the farm to be loaded into the truck it tries to verify whether it is positioned correctly, near other pallets that are to be loaded into the same truck. It does this by comparing its destination and contents with (the majority of) peer nodes on other pallets nearby. When a pallet is not positioned correctly or no peer nodes are found, it should raise an alert.

Next, the pallets are loaded into the truck transporting them to the harbor. Nodes can detect being loaded by 'hearing' another device, placed inside the truck, at which point they'll make the transition to stage 2. This device in the truck is programmed with its own itinerary, containing data about its identity, as well as the goods it is to be transporting. When in the truck, each pallet device requests from the truck device the company and truck IDs and records these into the log file.

While in the truck, pallet nodes do not have to verify anything, since no change in state will take place until they are taken out. They do have to detect being taken out of the truck, however, which can be concluded from absence of the truck, and presence of the wireless infrastructure (access point) of the harbor loading dock.

When unloaded on the dock, the sensor nodes again verify whether they are positioned correctly to be reloaded into shipping containers. The dock is equipped with electronic infrastructure capable of tracking each pallet's location, and based on this, each pallet verifies whether it is at the correct position. When placed incorrectly, it can directly send an alert message to the dock infrastructure that will inform workers to correct it.

For the last stage of the transport, the pallets are loaded into containers. These can be recognized by a matching shipping ID programmed into each container. Finally, when the container arrives in the distribution center, pallet nodes sense the distribution center access point and make the state transition.

3.4 Control of smart office spaces

Our last application scenario concerns a building control system for flexible offices. The system ensures energy preservation, while being hassle-free to use for office workers, and low maintenance for building managers.

This application scenario was previously published [EK09]. In this work we report on the integration of the work published previously with the Sensor-Scheme platform in Chapter 8.

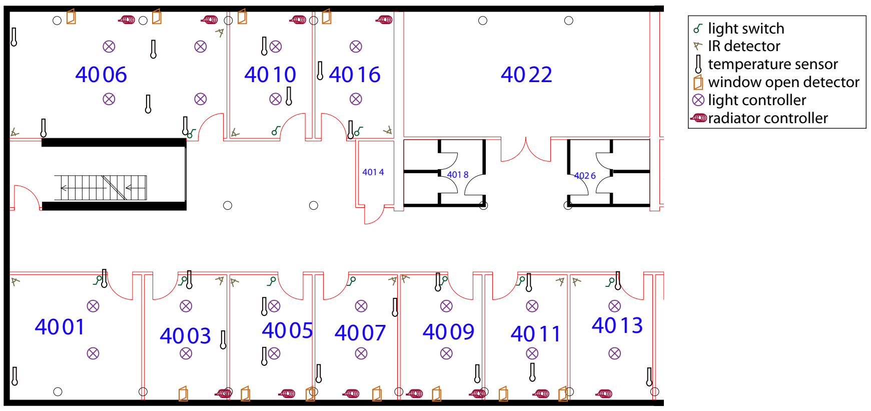


Figure 3.2: Floor plan for use in the scenario

Office rooms are fitted with a number of sensors and actuators that together control heating and ventilation, and switch lights. Figure 3.2 shows a floor plan of one of the floors of the University of Twente Computer Science building, with the positions of the various sensor and actuator devices shown. First, the ceiling-mounted lights in every room can be controlled with a wirelessly transmitting light switch device, usually placed near the room entrance. All lamps in the room contain a wireless receiver device that switches the lights on and off.

Furthermore, each room contains one or more central heating radiators, equipped with a controller device that controls the radiator valve. Rooms are fitted with several temperature sensors that together deliver a per-room average temperature. The desired temperature per room can be set, using a thermostat device in the room, which, using the room's average temperature, controls the radiator device.

When opening a window the radiators in the same room should be turned off, to preserve energy. To that effect, each window has a window open sensor attached. For additional energy preservation, presence detectors in every room monitor whether rooms are occupied, and if not, turn off the lights and heating.

Together, these devices make up a wireless sensor and actuator network. While the operation of every individual device is simple – a light switch only needs to send a message to the light controllers in the same room – configuration by hand of such a large network would be a complex and time consuming task: every individual device needs to be programmed with the identifiers of other devices it connects to. Changes to the configuration, for example when splitting a room or merging several rooms into a single large one, may involve multiple devices, that all need reconfiguration.

3.5 Requirements

Our aim is to define a software platform that is usable for all four of the described application scenarios, and – by extension – other WSN applications. This section derives the requirements for the platform posed by the application scenarios in this chapter.

3.5.1 Dynamic program loading

The most important requirement for our application scenarios is the ability to program and reprogram the devices after deployment, by transferring the packet across the wireless network. The different scenarios all rely on reprogramming in a unique way:

Intruder detection
While the intruder detection scenario does not specifically require the nodes to be reprogrammed while the application is operational, deploying a sensor network and developing the program may be more cost efficient when the application can be loaded after the network is deployed, and modified and reloaded of necessary.

Environmental monitoring
The environmental monitoring application requires to run a number of different sensor data queries simultaneously. When anomalous readings are encountered, a new query is sent into the network to gather more precise readings. The individual queries can be viewed as a kind of application, that needs to be programmed at the request of the user.

Logistics
The logistics scenario is in need for wireless reprogramming: for every shipment the appropriate parameters need to be programmed: the source, destination, and intermediate locations, content of the shipment and conditions that need to be guarded. Devices need to be reprogrammed for every subsequent transport, so reprogramming will occur frequently.

Smart office

The smart office network may operate uninterrupted and without reprograming during normal operation. When changes in the configuration occur, however, every device needs to be configured to properly connect to the other devices with which it needs to interact.

3.5.2 Dependent requirements

We have argued above that all four of our scenarios are in need of dynamic loading of their programs. As the state of the art indicates, there are several methods of achieving this, each with different properties and suggesting different uses. We now continue to analyze the applications' requirements for a program loading facility more closely.

Protection

Wirelessly loading a program onto the sensor nodes introduces a number of risks that nodes need to be protected from. Loaded programs may cause a node to be no longer accessible from the network. Such misbehaving programs can be created by developers making accidental mistakes – create bugs, or intentional – malicious programs such as computer viruses that have the purpose of gaining access to a device or damaging it.

This can be prevented by shielding application developers from the bare hardware resources such as peripheral devices and allowing access to only those memory regions in use by the application.

To our application scenarios protection is a crucial issue. For the logistics and smart office scenarios the users of the devices that need to program them may not be the owners of the hardware. The functionality that may be reprogrammed should thus be restricted to only application tasks. Hardware access and networking functions should not be reprogrammable to ensure the devices remain usable and wirelessly accessible. To a lesser extent, the environmental monitoring and intruder detection scenarios benefit from protection, as it protects the users from writing programs containing bugs that make the devices unusable.

Platform independence

Already a variety of WSN hardware platforms have been developed and more may be expected in the future. A software platform that supports to load applications on a variety of devices and let them cooperate in a single network will make sensor networks more usable. This is relevant especially for the smart office and logistics scenarios where devices from different manufacturers and multiple generations of hardware platforms need to cooperate. Similarly, for intruder detection and environmental monitoring, adding later generations of devices to the network at a later moment may be desirable.

Current devices differ in the CPUs in use, the amount of memory available, the wireless network interface and the peripheral devices such as sensors and actuators. Hardware independent software platforms must be able to execute a loaded program on each of those CPU architectures, use all the available memory, be independent of the hardware-specific networking protocols, and make use of the peripheral devices available.

Small program size

As argued above, the application functionality should be loadable on all devices of the network, and delivered through the wireless network. To be able to do this efficiently and with minimal energy consumption, it is important that the size of the program transported over the network is small. In wireless sensor networks communication is a limited resource, because the devices have low bit-rate radio's and the wireless medium is shared by a large number of devices. For example, the Deluge dissemination protocol [HC04] reports effective data rates of less than 90 bytes per second when disseminating data into all nodes of the network. The difference between applications of several hundreds of bytes and tens of thousands of bytes – the typical size of WSN binary images – is the difference between programming time of several seconds vs minutes. Energy use is also affected by the size of programs, as communication is responsible for the majority of energy consumed in WSN nodes.

3.5.3 Programming models

Independent of dynamic program loading ability, there is a large body of research into investigating programming models for sensor networks to ease the task of writing WSN applications. In the literature a number of methods have been proposed and analyzed, as described in Section 2.6.

Functional programming

A number of works propose the use of the functional programming paradigm. Regiment [NW04] is a lazy functional language akin to Haskell [Jon03]. Flask [MMWN07] is based on OCaml [OCa], and they share the vision to regard sensor data as a stream of data on which to apply pure functions. The functional programming paradigm also naturally supports aggregation of sensor values through the use of the *reduction* strategy.

Multi-threading

WSN operating systems are designed without support for multi-threading as a means to preserve memory. When using I/O operations, though, such as sending a network packet, the availability of multi-threading enhances programmability.

Communication

A significant part of WSN program's functionality revolves around communication. Programs may need to send and receive messages of various sizes and content. Certain tasks may require transmission of payload with variable size or structure, such as forwarding all data received from direct neighbors in the two-hop gossip protocol used for the intruder detection scenario, or the query results in the environmental monitoring scenario. The total message payload in such cases may exceed the size of a single packet, requiring some mechanism to split a single message into multiple packets, an recombine them upon arrival. Providing powerful communication abstractions simplifies construction of sensor network applications, reduces program size and enables construction of complex but efficient communication protocols.

Macro-programming

A research direction that has received attention in particular is macro-programming, to use a single program to specify the collective behavior of the entire sensor network. The state of the art (Section 2.6.3) proposes several macro-programming methods.

These programing tools and abstractions all have the goal of reducing the effort to build WSN applications. Tools and abstractions that are a natural fit to a particular application domain facilitate in creating short programs that reduce the chances for programming errors – bugs – and reduce development time. Shorter programs also benefit the resource-scarcity of sensor nodes, as we have argued above.

3.6 Conclusion

In this chapter we have defined a set of application scenarios that capture the broad area of uses for wireless sensor network technology. From these application scenarios we have derived a number of requirements for a software platform to implement them.

The following chapters will discuss the design tradeoffs for achieving these requirements on low power sensor nodes, the design and implementation of our platform, SensorScheme, and implementations of the scenarios using the SensorScheme platform.

Chapter 4

Design tradeoffs

The previous chapter has described four application scenarios for wireless sensor networks, and identified a number of requirements posed to a software platform on which to implement these applications. This chapter identifies the design tradeoffs for achieving these requirements.

We have identified two main requirements that are necessary for building these applications, but are challenging to build given the resource-restricted devices in use. In particular, wireless reprogramming is a challenging task resulting from the memory organization of sensor nodes: severely limited working memory (RAM), significantly smaller than program flash memory. Subsequently, low data rate radio's and dense networks result in slow transfer speeds of programs into the network.

Additionally, effectively building sensor network programs to load onto the network requires abstractions that may themselves be costly to implement, as they require significant amounts of working memory. Effective platform design involves decisions on what techniques and abstractions to use, and how to allocate memory for each, to keep memory use within limits.

In the design of any wireless sensor network software platform certain decisions have to be made trading off functional and nonfunctional aspects. functional aspects like program execution speed and memory available for allocation by applications are off-set with non-functional properties like peripheral and memory protection, and the risk of failing applications due to memory fragmentation.

4.1 Operating Systems

Before we set out to discuss the tradeoffs involved in the design of the SensorScheme platform, this section first lays out some of the characteristics of wireless sensor network operating systems. This will help us identify the problems and opportunities we describe in the subsequent sections.

In Chapter 2 we have discussed some of the operating systems developed especially for wireless sensor networks. The design of these differs considerably from desktop and server operating systems. The operating system represents the core functionality of a computer, managing use of and access to the computational resources – CPU time and memory – and peripherals.

4.1.1 Desktop operating systems

On desktop and server class devices, when the computer starts up, it loads a *kernel image* that is in control of special protection and memory management hardware in desktop and server processors. The kernel can load user applications, each in a private address space, effectively shielding off individual applications from each other's memory. When an application wants to access peripheral devices to read or write files on disk, communicate over the network, or display information on the screen, it requests to perform the particular operation to the kernel, which will execute it if permitted.

This separation between applications and the kernel is attractive because it allows the computer to load programs that may not be entirely trustworthy or error-free. All user applications have a private address space and are not capable of reading or writing any memory besides their own. When an application tries to access memory that it has not allocated, the kernel and other applications are not affected, and continue to operate normally.

Similarly, because the kernel controls access to peripheral devices, it can impose restrictions on its use. This prevents programs without the proper credentials to cause harm by improper use of peripherals, such as deleting data off the hard disk, crash or reboot a computer, or any kind of other unwanted behavior.

4.1.2 WSN operating systems

In comparison, embedded systems contain a single kernel image containing both operating system functionality and user application. The benefit of this approach to embedded systems is that the kernel image may be smaller and using less memory, since only those parts of the kernel and device drivers that are actually used by the application need to be in the kernel image. In addition, the simpler processors in use in WSN hardware platforms do not contain memory management and protection functionality and process isolation as provided by desktop operating systems is therefore impossible.

4.2 Memory allocation

Applications allocate memory in a variety of methods:

Global
Variables can be reserved as *global memory*, which makes it available to all parts of the application for the entire duration of the application. The compiler will allocate a memory region for these variables at a fixed address. In WSN operating systems, global variables are allocated at a fixed location in memory and cannot change in size. If the globally allocated data is not actually used it will still consume a fixed amount of memory.

Local
For variables local to individual functions, memory needs to be reserved only for the duration that the function is executing. These variables are assigned memory on the stack. Since a function *f* can call another function *g* that will execute within the execution time of function *f*, the stack has a last-in-first-out structure. The compiler makes sure that functions allocate and release their local variables on the stack automatically.

Dynamic
The third method of memory allocation is *dynamic allocation*. Dynamically allocated memory reserves contiguous chunks of memory from the *heap*. Using dynamic allocation programs need to explicitly allocate memory, and deallocate it later when no longer needed. Dynamically allocated memory can be used in several ways:

- When a value needs to be available for longer than the duration of a function. A common use is for values that functions return to the caller.
- To allocate multiple instances of a data structure when the number of instances is not known at compile time. WSN nodes may need to keep multiple messages in memory, for reasons of caching or while forwarding received messages to other nodes.

- To allocate arrays, the size of which is not known at compile time or will change over time. Many WSN applications need use an array to record data about neighbors the number of which varies over time. When the number of items in the array changes, the array size may need to change by reallocating a differently-sized array and copying the old contents.

While some WSN operating systems do make use of heap-stored dynamic memory allocation, its use is not common. Two of the most widely used operating systems, TinyOS and Contiki do not make use of it. One of the reasons (as stated by Gay *et al.* [GLvB$^+$03]) is that the small heaps used (just a few KB) are in risk of heap fragmentation.

4.2.1 Fragmentation

Memory fragmentation occurs in long-running programs (such as used in WSNs) when allocated memory regions end up being separated by many small unused memory regions. The problem in this situation is that it may become impossible to allocate memory, even though there may still be sufficient unused memory; the unused memory is not contiguous, but spread over many small regions, none of which is large enough to allocate the currently requested size.

In wireless sensor network platforms this may easily prevent large objects like packet buffers to be allocated. Packet buffers are relatively large memory objects, of approximately 100 bytes. In a heap of 4 KB, unfortunate placement of as little as 40 other objects may prevent any packet buffer from being allocated. Most of the heap may still be free in regions of less than 100 bytes. Such a situation may prevent a sensor node from sending or receiving any packets.

4.2.2 Non-fragmenting heap

An alternative method of dynamic memory allocation without fragmentation is used by Contiki: Allocated blocks of memory are moved to a different location in the heap when necessary to keep a single contiguous block of free memory. Unfortunately, when the heap-allocated block moves, pointers to the block in use by the application do not move. This technique can be made usable by using double indirection pointers: heap block pointers used by applications all point to a fixed-size *heap-pointer* data structure containing a pointer to the actual location of the memory block. When the block is moved, only the pointer in the *heap-pointer* needs to be modified to point to the new location. Applications using this method always need to perform a double pointer indirection to access the memory block.

This method effectively solves heap fragmentation at the expense of increased computation time due to double indirection and heap reorganization, and increased memory use due to storage of an extra pointer per heap block. The Contiki implementation uses heap-pointers of 6 bytes of size. Especially when large numbers of small objects need to be allocated, allocation of these six additional bytes may not be desirable.

4.2.3 Pool allocation

For long-running WSN applications fragmentation is a real risk that can be avoided by using only a more restricted form of memory allocation, which we will refer to as *pool allocation*. For data types of which multiple instances may be needed, such as packet buffers or neighbor information structures, arrays of multiple such objects are statically reserved in global memory. Programs can allocate multiple objects of a particular type from a pool, and need to explicitly deallocate each object. For every data type which requires allocation of multiple instances, a separate pool is reserved. The TinyOS and Contiki WSN OSes provide the *Pool* interface resp. *memory blocks* library.

While pool allocation precludes the occurrence of fragmentation and allows for a limited form of dynamic memory allocation, it does not serve every application need for dynamic allocation.

Applications may be using multiple pools, for different data types. The size of every pool needs to be determined at compile time, depending on the expected use of the associated data type. Pools of too large size waste memory, while too small sizes may harm application performance.

Additionally, when allocating arrays of sizes unknown at compile time (the third kind of dynamic allocation described above), pool allocation of fixed-sized objects presents no real solution.

4.3 Reprogramming

There are different methods of realizing reprogrammability, but not all of them are equally suitable for the purposes of the example applications that we target. This section discusses in detail the properties of several methods proposed in the literature and analyzes their applicability.

4.3.1 Whole image reprogramming

The simplest method to wirelessly reprogram sensor nodes is by replacing the current code image with a new one that is received over the wireless medium.

Procedure

Whereas embedded micro-controllers used in current WSN platforms have built-in facilities to reprogram them using a serial or USB connection to a desktop or laptop computer, wireless reprogramming requires software assistance.

First, the node must be able to receive network packets that contain parts of the new image. The node must store these parts in some kind of intermediate storage until the entire image is received. As internal RAM is not sufficient to hold a code image in these devices, an external RAM or Flash chip is used as intermediate storage. After the entire image is received, the node can replace the current code image in internal Flash by the received image. During this process no other tasks can be performed. Finally, when the code image is replaced the nodes can reboot itself, and start executing the new code image. After reprogramming, all application state is lost and a device needs to reinitialize its application, including discovering the network and its neighbors, before it can communicate again.

In case the image is somehow incorrect or contains bugs, it could be the case that the node is not able any longer to correctly execute this procedure again, leaving the node essentially unusable. To remedy this, some implementations load a basic 'safe' image into their external Flash that can be loaded when the device detects that it is malfunctioning. A watchdog timer (or grenade timer as described in the ExScal experiment [ARE$^+$05]) is present to facilitate this detection.

A *watchdog timer* is a special timer device integrated in the node's micro-controller that, once activated, counts down to 0 at which point it resets the CPU, unless it is restarted before the count down is finished. By periodically restarting the timer a reset can be prevented. Upon a reset from the watchdog timer the 'safe' code image will be loaded and executed.

The use of a watchdog timer cannot, however, protect against loading and execution of well-crafted, malicious code images [FC08]. The watchdog is started as part of the initialization of the code image and may not be activated by malicious images.

Execution

Compared to other methods, described below, execution of single image WSN applications is naturally the most efficient method. Programs are able to use all of the device's resources in the most efficient way possible. An application has the entire working memory available to it, sharing it only with operating system data structures. Application code is statically linked with the operating system functionality, which allows compilers to optimize both, and include only those operating system functions that are actually used by the application. The only restriction imposed upon an application is that it includes the reprogramming and code image replacement protocol into the code image.

Dissemination

Transmission of a complete program image may be an expensive undertaking. Image sizes range up to 128 KB, the Flash memory size of some of the low power sensor nodes. The *Deluge* dissemination protocol [HC04] reports a transmission speed of close to 90 bytes/second to reprogram an entire network. With this speed, a reprogramming operation can take as much as 25 minutes to finish for a 128 KB image.

Simply transmitting the entire image into the network also requires considerable communication, thus draining the nodes' batteries. Subsequent versions of code images contain at least partly the same code – that of the operating system – and in particular cases may be almost completely equal, such as when rolling out minor bug fixes or parameter updates. This suggests that it must be possible to transmit less data than the entire image to obtain a new version, by transmitting only the difference. Several authors propose such a method, based on a *diff*-like algorithm (Reijers and Langendoen [RL03], Stathopoulos *et al.* [SHE03]), based on the RSYNC protocol (Jeong and Culler [JC04]). These more efficient protocols add complexity to the reprogramming algorithm, resulting in increased size of the code images by several kilobytes and use extra RAM. Marrón *et al.* [MGL$^+$06] present a cost-benefit study of some of these protocols.

4.3.2 Loadable modules

A somewhat different method to allow programs to be loaded wirelessly is to enable loading *modules* of some sort: fragments of native code that can be loaded into a device's internal Flash besides a permanent kernel image. The WSN operating system Contiki [DGV04] optionally supports loading of modules, and SOS [HKS$^+$05] is designed especially to allow for loadable modules efficiently. TinyCubus [MLM$^+$05] enables the use of loadable modules for TinyOS.

Procedure

An advantage over whole image replacement is that when using modules it is possible to load several modules simultaneously, which can each be added, removed or replaced independently. For incremental updates involving only minor code changes, replacing only a single module out of several significantly reduces the amount of data to be transported to a node: only a single node instead of the entire image.

Costs

The advantages of using loadable modules come at a price, however. First, a module loading system requires a permanent kernel image which is responsible for management (such as loading and removing) of modules, and control access to the hardware resources that may be shared by these modules. Such a kernel image must contain functions that may not be used by any module but still occupy Flash memory. Management of modules also consumes working memory: modules must register themselves to the kernel to receive events such as arrived network messages and timer events etc. The kernel must keep track of these registrations.

Loading modules into internal Flash is another source of complexity. Modules are contiguous pieces of code that must be loaded somewhere into the internal flash. Because multiple modules may be loaded at the same time, the address at which to load it may differ. Modules must be compiled to use location independent instructions only, i.e. instructions should not contain hardcoded addresses of jumps, calls and memory locations, but only relative offsets from the current instruction location. Location independent code executes somewhat slower compared to when using hardcoded addresses.

Modules must be able to call functions made available by the kernel, which may be another source of execution inefficiency. It is possible to hard-code locations of kernel functions in module code, but this makes it necessary to recompile all modules after a kernel recompilation to make sure the addresses are still correct. Alternatively, some systems use indirect calls to access kernel functions, at a loss of execution speed.

Modules must share working memory as well, creating additional complexity.

SOS allows the use of 'standard' C library `malloc()` to dynamically allocate memory. This adds an allocation overhead of a few bytes per allocated memory block, and creates the risk of fragmentation, at which point the devices require a reboot.

Protection

Similar to whole image reprogramming, modules execute native code and therefore no form of protection is present, either from accidental behavior caused by programming bugs or from malicious programs. Modules may even corrupt each other's memory and cause each other's failure. Furthermore, module programmers do not control the size of the stack, which may again cause memory corruption in case modules consume more stack than available, for example when using nested functions.

Dissemination

Transmission of individual modules may proceed in a similar fashion to whole images. Individual modules can be transmitted using a protocol like Deluge [HC04]. Alternatively, FlexCup [MGL$^+$06] aims to reduce the data transmission of modules by using a differential method.

4.3.3 Interpreted programs

A third method of achieving the goal of reprogramming sensor nodes wirelessly is by using a custom-designed program representation which is executed on the nodes using a program interpreter. In general, this method shifts the balance between benefits and drawbacks significantly, and a wide variety of different methods are applicable. Some WSN systems using this method on low power sensor nodes are Maté [LC02], the Darjeeling [BLC09] and Perk [Cor08] Java VMs; on high speed sensor nodes Sensorware [BHS03] and Sun's Spots [Pro09], also discussed in Section 2.6.2.

Execution

Program interpreters use a custom program representation to encode their programs instead of the CPU's native instruction set. This is a widely practiced method outside the WSN world. Two of the most popular programming languages at the time of this writing, Java and C#, use this technique, as well as a large number of high-level or scripting languages such as Python, Perl, TCL and PHP.

The primary disadvantage of using program interpreters is the speed of execution. Lower execution speeds of between 10× to 100× compared to native processor instructions are typical, although reliable execution slowdown figures are hard to come by. In Chapter 9 we present some execution speed numbers obtained on the SensorScheme program interpreter.

For the Java and C# platforms, the slower execution speed has been significantly reduced with the use of Just In Time (JIT) compilers. A JIT compiler is a complex and large piece of software, however, that might not fit into the memory-constained devices that we focus on. Therefore, for WSN platforms we assume the use of interpreters without JIT compilers only. The interpreters for WSN platforms developed to the day of this writing [LC02, FRL05, Pro09, Cor08, BLC09, EHK$^+$07e] do not make use of JIT compilers.

Slower execution on the low speed CPU's present in WSN platforms may slow down execution of programs to consuming all of the processors' execution time while not meeting the real-time demands of the executing application. However, typical WSN applications have a low duty cycle to preserve energy.

Longer computation times will increase energy use. These effects can be expected not to be very large, though, and decreasing in the future, as argued in Chapter 2. Chapter 9 evaluates he energy used as a result of slower program execution in SensorScheme.

Program Size

The significant drawback of execution speed is balanced by program interpreters' powerful capabilities. Essentially, an interpreter or *virtual machine* can be seen as a 'virtual' CPU with a unique instruction set. But instead of transistors, the 'virtual' instructions are executed by software emulating their behavior.

Program interpreters or virtual machines provide the opportunity to design an instruction set possessing exactly the desired properties and behavior, without the burden of undesired effects. Desired properties of wireless sensor networks such as protection and memory efficiency can be realized by designing the interpreter to meet these goals.

As an example we take a look at the Maté virtual machine [LC02]. Maté is an interpreter for wireless sensor networks. It is a stack-based virtual machine, designed with the requirements of WSNs in mind, such as memory and execution efficiency. The instruction set consists of mostly single byte instructions, referred to as *bytecode*, analagous to the Java VM instruction set. Only few operations are encoded as instructions – arithmetic operations, variable and stack manipulations (eg. load, store, push and pop) and conditional and unconditional jumps. The majority of the 256 possible instructions encodes for common operations such as loading a constant, with different constant values.

Additionally, some instructions perform high-level operations such as reading a sensor or sending a packet. The second generation [LGC05] enables *application-specific VMs*, to include a choice of custom (high-level) instructions to be compiled into the instruction set. Such additional instructions can perform arbitrarily complex operations, including access to communication protocols like *abstract regions* [WM04], as Levis *et al.* show [LC02]. (Using compression techniques it is even possible to automatically generate virtual machines with individual instructions of less than a single byte, as Latendresse [Lat00] and Evans and Fraser [EF01] have demonstrated).

Programs for such a virtual machine may be very short – a single byte per instruction or less, and complex tasks such as inter-node collaboration can be encoded as a single instruction. Small, but still useful, programs can be encoded in well below a hundred bytes, as a 71 byte example program shows [LC02]. Transporting programs of such size is fast and efficient, even for wireless sensor networks' extreme low data throughput rate.

Our work, described in chapters 6 and 7 uses a somewhat different but still efficient method to represent and execute programs, and uses a compression scheme to further reduce its size during wireless transmission of program code. Chapter 9 evaluates and compares the program sizes using our interpreter and others.

Protection

Perhaps even more important than the small program sizes is the ability of interpreters to host a secure environment that protects WSN nodes from buggy or malicious programs. For programs transported and executed as native code protection is not possible. Such programs are able to read from and write to any memory location on a device, which is enough to take control of a device by malicious programs. Accidentally accessing – especially writing – the wrong memory location is a common cause of programming bugs, and may render a device unusable.

Interpreters eliminate this risk by allowing programs to access only those parts of memory under control of the interpreter, and only if programs are properly typed. Values stored in memory locations are all associated with a data type, and reading or writing may only occur if the source and destination types match. This is important especially when values are references to other memory locations, since only references to controlled memory locations may be stored. This is a long known and often used memory protection method, used as early as 1960 in the first Lisp implementation by McCarthy [McC62].

Protection against spurious memory access requires the use of a *type system* as part of the interpreter design and primitive operations or instructions to load and store typed values in a protected manner. Section 6.4 describes SensorScheme's type system.

Platform independence

Another distinguishing feature that program interpreters have is platform independence. Programs run on devices with different CPU architectures if implementations of the program interpreter exist for these architectures. In the face of a need for compatibility with future devices and implementations, as is particularly the case with the smart office and logistics scenarios, platform independence leaves more freedom to the design of future devices.

Dissemination

In the current implementations, the dissemination method for program interpreters is tightly connected to the design of the interpreter itself. Maté uses a variant of the Trickle [LPCS04] protocol to disseminate programs as a number of independent parts or *context*s, each with a maximum size determined by the packet size in use by the communication's implementation. SensorWare [BHS03] is designed to distribute programs as part of a distributed agent architecture: The agent program on a node issues a command to distribute the program to other nodes within direct communications reach. Similarly, Agilla [FRL05], while based on the Maté virtual machine uses an agent abstraction to distribute its programs. SensorScheme uses a unique approach, described in Chapter 7.

4.4 Threads and events

Within the wireless sensor network operating systems developed to this day, the use of multi-threading has been an important and motivating design choice. Operating systems like MANTIS [BCD+05] are created especially to include multi-threading, while TinyOS [HSW+00] and Contiki [DGV04] have deliberately left multithreading out.

The *TinyOS* WSN operating system is programmed using the *nesC* programming language, a extension to C for *n*etworked *e*mbedded *s*ystems. NesC extends C with a component system suited specifically to a hardware-bound kind of programming, where a nesC component matches a hardware component like a chip on the sensor node device or a part of it. The component uses interface definitions to separate specifications from implementations. This model allows for code reuse of hardware component drivers on different platforms, such that for example the device drivers of the CC2420 radio chip can be used directly on both the MICAz and TMote platforms, that use different CPU's. NesC further has notions of *commands* and *events* that represent the bidirectional interaction between connected *components*.

The memory limitations of WSN platforms has motivated the operating system designers to use only a single call stack shared by all tasks. Operations that would cause a thread to block in multi-processing or multi-threading systems, such as performing input and output operations, need to be implemented in a split-phase manner in event-based systems.

4.4.1 Events

Event-based systems execute *tasks*, that run to completion before the next task is run. Tasks can be scheduled to run by interrupt handlers or by tasks themselves or other tasks. When performing input and output operations that may take some time, the task should not wait until completion of the operation, but finish immediately, to allow other tasks to execute. When the I/O operation is finished, it signals an event that will resume the program that performed the I/O operation.

As an example of a split-phase operation we take sending a message, shown in Listing 4.1. The function (or *command* in TinyOS) `send(addr, msg, ...)` puts the message `mess` in the MAC layer transmission queue, schedules the transmit task and returns immediately. The task calling `send` should now finish, before the message will actually be sent in the transmit task. After the message is sent a callback function is called (or an *event* is signaled in TinyOS) reporting the error or success status of the transmission. In the event or callback the application continues its operation.

Using events, the logical flow of control of operations that occur before and after the I/O request has been separated into two individual functions. Moreover, in case the program contained live state at the time of the I/O operation, it must be saved manually, and retrieved after completion.

```
interface AMSend {
  command error_t send(am_addr_t addr, message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t error);
  ...
}

...

task void sendMsg() {
  ...
  error_t error = call AMSend.send(addr, msg, sizeof(my_msg_t));
  if (error != SUCCESS) {
    reportFailure(error);
  } else {
    return;
  }
}

event void AMSend.sendDone(message_t* msg, error_t error) {
  if (error != SUCCESS) {
    reportFailure(error);
  } else {
   /* continue computation */
  }
}
```

Listing 4.1: Split-phase send example

4.4.2 Multi-threading

Event-driven and multi-threading architectures are two equivalent approaches for concurrent program execution [LN79]. Still, multithreading systems provide a more natural programming experience, according to von Behren *et al.* [vBCB03], because of two major drawbacks of event-driven architectures – manual stack management and manual flow control. This deficiency of WSN architecures was already pointed out by Kasten and Römer [KR05] who note that "*many conceptual operations need to be split among multiple actions*", and "*programmers must include additional management code, which obscures the logical structure of the application and is an additional source of error.*" This has motivated a number of efforts to add various degrees of threading support for WSNs up until a fully preemptive multithreaded WSN operating system.

ProtoThreads [DSVA06], part of the Contiki [DGV04] OS are the simplest solution to adding thread support. ProtoThreads aims to remove the split-phase nature of an application's main loop by using a combination of C macro's. It does not allocate multiple thread stacks and protothreads do not retain local variables across blocking calls, and as such it is merely a source code refactoring technique.

Fibers are part of the abstract regions [WM04] communication abstraction for TinyOS. Fibers present a method of creating blocking calls in an event-driven operating system. Only a single blocking context may be present, allowing the blocking *application fiber* and the event-based *system fiber* to share a single stack. Context switching requires only to save and restore the CPU's register set.

Y-threads [NPR06] is a hybrid thread solution, providing preemptive multi-threading, while preserving memory. It distinguishes *Run-to-Completion Routines* from pre-emptable tasks, where tasks may invoke RCRs. Because only a single RCR may be active at any time, all RCRs share a single stack to reduce memory allocated to thread stacks.

TinyThread [MS06] is a thread library for TinyOS. It supports creation of multiple threads, each with their own stack, with optional preemption of threads by others. TinyThread provides a utility, 'stack-estimator' to calculate the minimum stack size needed for each thread to execute safely.

The BTnut [btn09] operating system implements cooperative multithreading. Each thread requires its own stack, and unlike TinyThread, the stack size for each thread needs to be estimated by the programmer himself.

The MANTIS [BCD$^+$05] OS is a fully preemptive multi-threaded operating system for embedded systems. The other methods do not provide pre-emption by default.

Having multiple threads of execution simultaneously active (multi-threading) requires memory allocated for each of the threads. Each thread is essentially a procedure call stack that captures the thread's current state of execution (possibly with other associated data).

Memory for the thread stacks may be allocated in a variety of ways. First, using pool allocation (sect. 4.2.3) a fixed number of thread stacks of fixed size may be allocated. However,in the face of dynamic reprogramming, the number and size of the thread stacks needed by an application cannot be known in advance.

The thread stacks may also be allocated as needed from the heap. Determining the maximum needed size of the stack is not possible in all cases; it is therefore unclear what size a stack should be allocated in its entirety. Alternatively, stack space may be allocated one call frame at a time as is the case in Darjeeling [BLC09]. This method ensures that stack memory consumption is only as much as is currently in use, not wasting stack space that is allocated, but not in use. In combination with a non-fragmenting heap of some sort, per-call-frame allocation is a safe and memory-efficient design for a multi-threading system. The method comes, however, at the expense of extra computation time caused by stack frame allocation every procedure call. The technique used by SensorScheme, described in Chapters 5 and 6 uses a similar method.

4.5 Communication

Network communication takes a very important role in WSN applications, partly because it is the major form of I/O for WSN nodes, and because of its high impact on the energy budget.

In contrast to desktop operating systems, however, communication abstractions in WSNs are meager. The model of communication offered by operating systems consists of sending individual small, fixed-sized packets. This makes applications highly dependent upon platform-specific details such as the packet size.

Facilities enjoyed in TCP/IP networks, like automatic packet (de)fragmentation and reliable (streaming) transmission are not available to WSN application programmers. Having a similar kind of communication service for WSN platforms could simplify writing applications, which are more portable across platforms using different packet sizes, and potentially reduce communication and energy use.

While implementations of TCP/IP (version 4) as well as IPv6 are available for WSN platforms [Dun03, HC08], for most WSN communication TCP/IP is not very suitable. It is designed for symmetric bidirectional peer-to-peer streams, and setting up a connection is more heavy-weight than suitable for WSN communication. In wireless sensor networks, communication patterns have an asymmetric one-to-many or many-to-one nature, and minimizing communication is crucial for energy preservation.

Current WSN network protocols fail to abstract from the physical packet size of the underlying radio technology. Network stack layers that take care of automatic packet (de)fragmentation aren't generally available. As a result, applications are tuned to transmit messages that fit well into the packet size of the hardware platform used. Such meticulous tuning is one of the reasons that sensor network application development is a highly skilled undertaking.

Our application scenarios demand an environment that is hardware platform-agnostic to the greatest extent possible. In the design of SensorScheme we focus on communicating messages that may be the size of multiple packets of the underlying protocol stack, without the need for programs to have been built with the knowledge of the packet sizes of these underlying protocols. To realize this, we need to separate the logical content of a communication, which we will refer to as a *message*, and the carrier of this content referred to as a *packet*, of which multiple may be needed to carry a single message.

As with other design tradeoffs, the limited memory in WSN platforms plays a decisive role in the design of a multi-packet communication method.

4.5.1 Example: gossip protocol

As an example, lets consider the intruder detection application scenario of Section 2.1.1. The data sent is a variable number of small, fixed sized data items. If all data items to transmit were located in an array, the entire array can be transmitted by splitting it up in packet-sized chunks and transmitting them one by one. The receiving nodes will then collect the transmitted packets and recreate a copy of the entire array in their local memory. When a node receives multiple messages in the same time (each from a different neighbor), each must be recreated separately on the receiving node.

One possible implementation strategy would be to define a maximum logical message size to a multiple of the packet payload size, and message buffers of the same size. The number of message buffers defined determines the maximum number of messages that can be received simultaneously. To make sure nodes can receive and later send all the data items intended for them one would have to define as many memory buffers of as large a size as possible. This reserves excessive amounts of memory. For a network with maximum degree of n (a node's degree is the number of nodes it can communicate with) n arrays of n data items each are required, resulting in memory allocation of $O(n^2)$, while for the majority of nodes, with degree $<= m$, for any $m < n$, only a fraction of memory equal to m^2/n^2 is used. This means that for a node with degree $m = n/2$, only $(n/2)^2/n^2 = 1/4$ of the reserved memory is ever used, and leaving 3/4 of the node's reserved memory unused.

Alternatively, to address the excessive memory reservation by explicitly limiting the number of buffers and their size, nodes will need to disregard messages received from some neighbors, as they lack available buffer space. As neither strategy yields acceptable implementations, existing WSN frameworks and abstractions have largely ignored the efficiency opportunity of packing related data into the smallest number of packets possible. Instead, common practice is to send a single data item per packet, so it can be processed immediately upon reception, and prevents the need for a large buffer space in memory.

In short, design of communication services independent of packet size is prone to excessive memory consumption, in a manner similar to other design choices discussed in this chapter. SensorScheme includes a novel communication abstraction that is conservative in its use of memory, described in Chapter 7.

4.6 Memory organization

In the preceding sections we have identified a number of design trade-offs relevant to our software platform. The memory allocation method available is a major influence on what programming tools and abstractions can be offered on low power sensor nodes. Similarly, dynamic program loading is bound to strong requirements towards memory organization and allocation.

The different structures that have to be allocated in memory are the following:

- multiple call stacks to support multithreading
- multiple partially received messages for multi-packet communication
- program code containing instructions and constants of unknown size
- application data including arrays of unknown size to store data received from neighbors

For each of these memory structures, allocating them in small sections rather than large monolithic memory regions minimizes memory fragmentation. Furthermore, this enables allocating as many of these structures as needed and of sizes as large as needed by dynamically loaded programs.

We have chosen to use a single pool of small, equally-sized *cells* as the only application-accessible memory. Cells may link to other cells, to create larger compound data structures. To enforce memory protection, the data types of values in the cells should be accessible to the interpreter. Automatic memory management in the form of *garbage collection* must make sure to automatically retrieve cells on longer in use. Chapter 6 describes SensorScheme's memory and type system, and its implementation on low power sensor nodes.

All of the dynamically loaded programs' required memory will be allocated as collections of linked cells from the cell pool, including its program code, call stack, packet buffers, and application-allocated memory.

By using a single source of memory to allocate program code, a call stack and application data, the program's complexity in terms of code size, call depth or memory use is not arbitrarily restricted. Instead, programs of broadly varying size, complexity and behavior are able to run on the same interpreter platform.

4.7 Programming Languages and paradigms

The WSN platform to be designed requires an effective programming language to build programs for wireless sensor networks. The effectiveness of programming languages in general is a topic about which many opinions exist, but little facts. Only a few scientific studies have been published in relation to this question, and most writings on this topic can be regarded as the informed opinion as voiced by experts.

4.7.1 Programming languages

Hattori *et al.* and Prechelt have published two similar studies [HKW85, Pre00] that compare the effectiveness of different programming languages for a particular programming task. Hattori compares Lisp, Prolog and Ada; Prechelt compares C, C++, Java, Perl, Python, Rexx, and Tcl. Gat [Gat00] additionally repeated Prechtelts study for Lisp. Both Hattori's and Prechtelt's studies let programmers of various skill levels program a fixed set of problems using one of the languages under review, and measure various quality aspects, such as time needed to perform the task, size of produced program, number of bugs in program, and performance characteristics of the program produces, such as execution time and memory use. Both studies show that the results vary widely, and do not show a significant advantage of any language over others. A different conclusion that is supported by these studies is that the size of the produced program is a good predictor for its quality: short programs contain fewer bugs, execute faster, consume less memory and have been written in shorter time than longer ones. Programming languages that appear to facilitate writing shorter programs also perform better on quality indicators.

While these studies are far from conclusive, it seems safe to conclude that writing short programs should be a goal to pursue for wireless sensor network systems. Short programs ensure fast and energy-efficient program transfer, and reduce memory consumption by program code. More importantly, it may be the key to absence or reduction of bugs, quick development time, fast execution speed and reduced memory consumption.

Both studies show that Lisp enables writing compact programs, that take little time to write and contain little bugs. The use of Lisp, or a descendant language such as Scheme for a WSN platform may bring these advantages to sensor network applications and their developers.

Two of the reasons Gat mentions in his study of Lisp for the reduced code size are dynamic typing and the use of powerful abstraction facilities like first-class functions. Dynamic typing as used in Lisp reduces code size by not including type declarations in program texts. Abstraction facilities such as higher order functions and closures enable powerful programming strategies able to reduce the size of source code. The next chapters will explore how these these features can be used to make concise and effective sensor network programs.

4.7.2 Programming paradigms

Besides focusing on the suitability of individual languages for a given problem domain, we can distinguish *programming paradigms* in which these languages fall. Van Roy [VR09] proposes a classification for programming paradigms. It distinguishes paradigms such as imperative, object-oriented, functional and logic programming. Individual programming languages realize one or more paradigms, which in turn is defined by a set of programming concepts, such as mutable state, higher order functions or unification (as used in logic programming).

Languages realize the concepts involved in different paradigms in a variety of ways: sometimes by little more than change in language syntax, as is the case of C vs. C++; sometimes a change in the use of computing resources is involved, as is the case with the inclusion of higher-order functions and closures.

According to Van Roy, *each paradigm supports a set of concepts that makes it the best for a certain kind of problem.* The kinds of problems encountered in wireless sensor networks applications determine the suitability of particular paradigms.

Research into the suitability of programming languages or paradigms for wireless sensor networks presents itself in the form of a number of proposals of programming languages and dialects for programming WSNs, especially those referred to as macro-programming (see Section 2.6.3).

Functional programming

Macro-programming using functional languages has been proposed as a suitable paradigm for sensor networks a number of times, for example in the form of the Regiment [NW04] and Flask [MMWN07] platforms. A prime distinguishing feature of functional languages is the use of higher order functions to iterate over sets of nodes and sensor values, using stateless processing of sensor data – a good match to at least a subset of WSN applications.

SensorScheme, as a descendant of one of the original functional programming languages – LISP, provides the benefits of functional programming to wireless sensor network programs. SensorScheme demonstrates an additional benefit of the use of functional-style programs through its method of program specialization by partial evaluation, discussed in Chapter 8.

Imperative programming

While a functional programming language is a good match to specify data processing in sensor networks, it cannot always adequately cope with access to peripherals and perform I/O actions. Imperative programming languages, such as C and nesC, or Java, Python and other languages designed for memory-safe execution environments, are better able to perform I/O, and manipulate hardware and memory state. Within wireless sensor networks, access to sensors and actuators, and I/O in the form of network communication are the basic ingredients of any application. Purely functional WSN platforms – of which Regiment is the prime example, have more limited methods of performing I/O and communication. The imperative paradigm is essential to a generic WSN platform intended for the wide range of uses that we propose in our example scenario's.

Other paradigms

Other programming paradigms have been applied to sensor network applications, in particular object-oriented programming in the form of Java. It is not exactly clear, however, that object-oriented languages features such as dynamic method dispatching and type inheritance are of real benefit to WSNs. Sensor network applications are inherently small in size, whereas object-oriented languages claim their benefits especially for large-scale programming efforts.

The use of Scheme as a WSN programming language is that for WSN applications where the need for object-oriented features does arise, these are accessible through an object library for Scheme, such as those proposed by Kiselyov [Kis99, Dic92].

Prolog-like declarative programming with inference has been proposed for WSNs in the RuleCaster [BK06] and Cooperative Artefacts [SGKK04] systems. Similarly, this functionality can be implemented within Scheme as Felleisen [Fel85] shows. This work does not discuss, however, the use of object-oriented or logic programming paradigms within the SensorScheme platform, but leave this as an exercise to the reader.

4.8 Conclusion and outlook

As with any other software platform or operating system, the design of a platform for WSNs requires the consideration of a wide variety of aspects of system organization and behavior, such as process and thread scheduling, handling of asynchronous events, memory allocation, and, especially important for WSNs, communication. Each of these aspects poses restrictions on the overall design of the system, and their role in the system should be considered in unison. This chapter has analyzed the alternatives and described the choices made for the SensorScheme platform.

Considering the requirements of efficient reprogrammability and protection posed by the application scenarios, we chose to use a program interpreter to execute WSN applications. Making the design as simple as possible, our platform will allocate the various in-memory structures such as program code, call stack, communication buffers and application data from uniform memory cells and use automatic memory management to ensure memory safety.

The choice of programing language for any piece of software is still guided not by any objective standard, but by opinion and experience. The following chapters describe the SensorScheme language and platform, which is designed to suit the design decisions laid out in this chapter. Chapter 5 introduces the SensorScheme language, and show by example the various techniques to write short and effective WSN applications. Chapter 6 discusses the overall design of SensorScheme and how it realizes the design goals we have discussed in this chapter.

Chapter 5

Programming techniques

After describing the application scenarios and broad design decisions this chapter focuses on this dissertation's goal of developing programming methods and abstractions to reduce the effort of programming WSN applications. This chapter demonstrates the use of a number of programming language concepts and abstractions and how they facilitate building WSN applications.

In particular, we will demonstrate the following concepts:

- closures
- higher-order functions
- reduction
- continuations
- dynamic loading and evaluation
- various uses of linked-list data structures

We will be using the SensorScheme language to create example implementations of the scenarios described in Chapter 3. This chapter aims to show that the use of these techniques, available in SensorScheme, leads to compact programs that are easy to build and understand.

5.1 SensorScheme

As the name suggests, SensorScheme is based on the Scheme [ADH$^+$98] programming language, and its ancestor Lisp. SensorScheme is not unique as an interpreter for the Scheme or Lisp languages for resource-constrained computers and embedded systems. Several other implementations exist, such as the BIT [DF05] and PICBIT [FD03] Scheme implementations for embedded processors. In fact, the first Lisp implementation [McC60] ran on a computer – the IBM 704 from 1955 – with similar amounts of memory (4 k or 32 k of 36 bit words of magnetic core memory) to WSN platforms and even slower computational speed (about 40 k instructions per second). While these implementations give good guidelines for implementing Scheme and Lisp on a resource-constrained computer, they are not designed for the type of devices in use in wireless sensor networks. Instead of working with textual input and output from terminals and files, sensor networks deal with packets from the wireless network, and sensors and LEDs as I/O devices.

We have chosen to design a WSN platform based on Scheme because Scheme provides programming techniques and concepts that are instrumental to reducing program size and complexity which are not available to WSN platforms that are built using procedural, imperative languages like C and its derived NesC. This chapter will introduce those techniques and show how they are applicable in WSN applications. These techniques are not new or unique to SensorScheme, but available as a result of SensorScheme's heritage of standard Scheme.

Additionally, this chapter introduces SensorScheme's contributions which consist of modifications and additions to make the Scheme language usable and efficient on wireless sensor networks. We will demonstrate the following SensorScheme contributions:

- a reduced set of data types to enable efficient data storage;
- input and output procedures to access WSN-specific peripheral devices;
- procedures to access the processor's scheduling and timing mechanisms;
- a communication mechanism tailored to the needs of wireless sensor networks.

Each section in this chapter discusses an example SensorScheme program or fragment, including example implementations of the application scenarios in Chapter 3. The above-mentioned programming techniques are discussed upon their first use in any of the discussed programs.

5.2 The basics

Before we describe programs to be written in SensorScheme, we discuss its basics of syntax and semantics. The following material is an introduction to the Scheme language and does not contain contributions specific to SensorScheme, except where noted. For a more extensive introduction into the Scheme language, the reader is referred to a Scheme course book [ASS96, Dyb09, Kri02] or the Scheme standard document [ADH+98].

5.2.1 Syntax

The SensorScheme syntax, equal to regular Scheme, is extremely simple: it consists of tokens surrounded by parentheses. Instead of parentheses, square brackets may be used, to enhance readability. Comments begin with a semicolon and continue until the end of the line.

```
(+ 1 2 3 4) ; this is a comment
[+ 1 2 3 4] ; this line is equivalent to the line above
```

5.2.2 Expressions

SensorScheme programs consist entirely of expressions. Expressions are written in prefix notation enclosed within parentheses. Unlike many other languages, the parentheses do not play a role in denoting operator precedence, but are part of the expression syntax. Adding or removing parentheses around an expression changes its meaning. SensorScheme does not distinguish statements from expressions, and does not make use of infix operators.

When evaluated, an expression always returns or *evaluates to* a value. In these examples we use a > prompt in front of evaluated expressions, and their result preceded by --> on the following line.

```
> (+ 1 2 3 4)
--> 10

> (> 5 2)
--> #t ; true
```

5.2.3 Variables

Special expressions are used to define and access variables. A **define** expression allocates a new global variable and binds a value to it. A variable reference returns the currently bound value. A **set!** expression binds a new value to an existing variable.

```
> (define a-var 8)
> a-var
--> 8

> (set! a-var 2)
> a-var
--> 2
```

Notice that the name of the variable `a-var` above contains the non-alpha-numeric character '-'. In fact all identifiers (called *symbols* in Scheme) may contain a variety of characters reserved in other languages for operators and other syntactic elements. The `+` in the first example also is a regular Scheme symbol.

5.2.4 Values

The values returned from expressions may take several forms. These include integer numbers and boolean values **#t** and **#f**. Symbols, which were used as variable references, are valid SensorScheme values as well. The **quote** expression has a single argument, which it returns unaltered, as shown below. A preceding quote character '`'`' is used as a shortcut notation for the **quote** expression.

```
> (quote a-var)
--> a-var
> 'a-var
--> a-var
```

Besides these atomic values, Scheme uses more complex values such as lists, displayed as a sequence of values enclosed by parentheses.

```
> (list 1 2 (+ 1 2))
--> (1 2 3)
```

Note that the syntax of lists is similar to the expression syntax. In fact, expressions are a special kind of value that may be *evaluated.* In the above example `(list 1 2 (+ 1 2))` is an expression that uses the `list` operator to return a list value, written as `(1 2 3)`.

The property that programs in a language are a special case of values native to the language, called *homoiconicity*, is essential for some of the ways SensorScheme can be used on sensor networks, as Section 5.8 describes.

Programs may also contain literal list values as an argument to **quote**, in which case the literal list is returned from the **quote** expression.

```
> (quote (1 2 (+ 1 2)))
; equivalent to '(1 2 (+ 1 2))
--> (1 2 (+ 1 2))
```

In Scheme, lists are implemented as linked lists consisting of *pairs* and (created with the `cons` procedure) and terminated with an *empty list* `()`. A pair stores two values, which can be retrieved with procedures `car` and `cdr`:

```
> (define ls (cons 1 (cons 2 ())))
> ls
--> (1 2)

> (car ls)
--> 1
> (cdr ls)
--> (2)
```

5.2.5 Conditionals

One kind of expression is the conditional. It starts with the keyword **`if`** , followed by three expressions: a predicate expression, a ‘then’ expression and an ‘else’ expression. The conditional expression returns either the result of the ‘then’ expression or the ‘else’ expression, depending on the result of the predicate.

```
> (define a 1)
> (define b 2)
> (if (> a b) a b)
--> 2
```

The ‘else’ expression is optional. When omitted, a conditional expression returns *false* in case the predicate evaluates to *false*.

5.2.6 Procedures

In Scheme functions are defined using the **`lambda`** expression. The keyword **`lambda`** is followed by a list of parameters (zero or more) and a body, consisting of one or more expressions. Section 5.2.8 explains more about procedure bodies.

```
> (lambda (x) (* 2 x))
--> #<procedure>

> (define double (lambda (x) (* 2 x)))
```

The **`lambda`** expression above has only one argument named `x` and a single body expression. The result of a lambda expression, a *procedure* value, may be bound to a variable to create a named global function.

Calling the defined function works using the general expression form: a sequence of sub-expressions between parentheses, where the first one evaluates to the procedure value to be called.

```
> (double 3)
--> 6
```

Instead of the function's name the first subexpression in a call may be any expression that returns a procedure value. Like the expression below, a **lambda** expression is a valid procedure to call in a call expression.

```
> ((lambda (x) (* 2 x)) 3)
--> 6
```

Naturally, the arguments to procedure calls may themselves be expressions instead of only literal values. These subexpressions return values that serve as the arguments in the procedure call.

```
> (double (double 3))
--> 12
```

In fact, what happens when evaluating a procedure call, is that each of the expressions in the call expression (including the first, the function 'name') are evaluated in turn; the value returned by the first expression must be a procedure value. Upon calling this procedure, the remaining subexpressions' values are bound to the **lambda** expression's parameters. Within the procedure's body, `(* 2 x)` in the above procedure, a variable reference to `x` reveals the value bound to it.

5.2.7 Let

Local variables are created using **let**. A **let** expression contains a list of variables and values followed by a body:

```
> (let ([a 2]
        [b (* 2 5)])
    (+ a b))
--> 12
```

The variables and values list consists of pairs of a variable name and an expression that returns the value to be bound to the variable. The **let** body consists of one or more expressions that are evaluated in sequence. the last body expression's return value is also the value returned by the **let** expression.

The local variables introduced by the **let** expression are accessible to its body expressions. If any of its body expressions is itself a **let** expression, the variables of the outer **let** are also accessible to the inner **let**:

```
> (let ([a 2])
    (let ([b (* a 5)])
      (+ a b)))
--> 12
```

5.2.8 Side effects

Instead of only using a single body expression, multiple expressions can be used. While the results of these expressions are lost, using multiple expressions is useful to perform operations that have side effects. For example, performing I/O operations, or changing the value of a variable with **`set!`**.

```
> (let ([a 2])
    (print a)
    (set! a (+ a 2))
    (let ([b (* a 5)])
      (print b)
      (+ a b)))
--> : 2
    : 20
    24
```

Procedure `print` in the above example prints the values given as its arguments to the device's serial port, shown as lines starting with :. It is evaluated only for its side effects.

5.2.9 Variables and scope

SensorScheme distinguishes between global and local variables. Both are accessed and assigned using the same syntax. There is a difference however between the two. Whereas global variables are accessible from any part of the program, local variables are accessible only in the *lexical scope* where they have been defined. For variables introduced in a **`let`** expression their lexical scope contains only expressions within the **`let`** body of their definition, and all expressions recursively defined within those expressions. Access of a variable outside of its scope will result in an error:

```
> (let ([v 1])
    (print v)) ; within the scope of variable v
  (print v) ; outside of the lexical scope

--> : 1
    error
```

Whenever there exist multiple variables with the same name, the innermost local variable takes precedence:

```
> (define v 1); this is a global variable
  (let ([v 2]) ; the outer local variable
    (let ([v 3]) ; the inner variable
      (print v))
    (print v))
  (print v)

--> : 3
    : 2
    : 1
```

5.2.10 Macros

The **let** expressions can be rewritten using lambda, for example:

```
(let ([a 2]
      [b 10])
  (+ a b))
```

is equivalent to

```
((lambda (a b) (+ a b)) 2 10)
```

The Lisp and Scheme languages use *macros* to transform the syntax of expressions into others. We will not detail the language used to define macro's; instead we only define somewhat informally the syntax of macros and the result of transformation. Appendix C.3 describes all macros used in this work. We write macro transformations as an expression followed by a line ==> and the transformed expression.

Besides the **let** macro the Scheme standard defines several other forms, including **let***, and **letrec**. In SensorScheme these are also implemented as macros, and defined in appendix C.3. Macro **let*** allows the values assigned to variables to refer to previous variables in the same **let*** definition, and **letrec** is commonly used to define recursive functions that are bound to a local variable.

Another commonly used macro is the **begin** macro:

```
(begin
  (print a)
  a)

==>

((lambda ()
   (print a)
   a))
```

It is used when multiple expressions are needed in places that may contain only a single one, such as in **if** expressions or function arguments:

```
(if (> a b)
    (begin
      (set! b a)
      a)
    (begin
      (set! a b)
      b))
```

A **define** macro for definition of global functions is used to create the standard Scheme shorthand definition:

```
(define (max a b)
  (if (> a b) a b))

==>

(define max (lambda (a b)
                (if (> a b) a b)))
```

All other macro's used in the programs in this chapter are listed in Listing C.3.

5.3 A first program

The first SensorScheme program that we will discuss here introduces some methods unique to SensorScheme to build wireless sensor network programs. It showcases the typical program structure in the form of a delay loop and method of communication for SensorScheme programs. It is a simple program, similar to the TinyOS `RadioSenseToLeds` example application.

Time and scheduling

The `sensor-blink` example program, shown in Listing 5.1 periodically sends a message containing a counter value. Nodes receiving such a message will call `blink` on line 5 which displays the lower bits of the counter value on the nodes' LEDs. The `blink` procedure is a *primitive procedure* that interacts with the operating system to control the LED peripherals.

The program is contained in a SensorScheme module definition (line 1), that imports all the provided definitions from module `thesis-base` (line 2). The full source of module `thesis-base` is available in Listing C.1.

The program itself consists of two definitions and an initialization expression. The **`define-handler`** expression (line 4) defines a message handler. We will explain its use in the next section.

```
   (ssmodule sensor-blink
2    (require "thesis-base.ss")

4    (define-handler (sensor-blink-msg n)
       (blink n))
6
     (define (time-loop t)
8      (call-at-time (+ t 4) time-loop)
       (bcast (msg sensor-blink-msg (sense-temp))))
10
     (time-loop (now)))
```

Listing 5.1: sensor-blink example program.

Procedure `time-loop` is defined on line 7. It has a single parameter `t`. The first body expression of `time-loop` (line 8) calls `call-at-time`, a SensorScheme-specific primitive procedure that schedules a function to be called at time `t` $+$ 4 timer ticks. The call to `call-at-time` immediately returns and execution continues with the next body expression at line 9.

When the scheduled time arrives, the procedure `time-loop` will be called taking as its single argument the time it was scheduled, which is the current time at the moment of calling `time-loop`. Procedure `time-loop` again schedules the procedure to be called at a later time, creating an infinite loop called at regular intervals.

At the start of the program the initialization expression on line 11 starts this loop by calling `time-loop` with `(now)`, the current time, as argument. `now` is another primitive procedure unique to SensorScheme that returns the current time reported by the OS.

Time is measured in SensorScheme at 1/16 second intervals. The current time, as produced with the function `now` is the number of $1/16s$ intervals since the device was turned on. This results in `time-loop` being called 4 times per second in the program.

Communication

Communication in SensorScheme takes place by calling any of a number of primitive procedures that take as argument a *message* and send it using the operating system's communication facilities. Procedure `time-loop`'s second body expression (line 9) calls communication primitive `bcast` to broadcast a message. The message itself is constructed using macro **`msg`**. Different types of messages are identified by a message type symbol – `sensor-blink-msg`. The message's content is a single sensor value, obtained from the device's temperature sensor, which can be read by calling the primitive procedure `sense-temp`.

When a message arrives, SensorScheme uses the message type symbol to look up a *message handler* of the same name, i.e.. `sensor-blink-msg`, defined at Listing 5.1 line 4. The **`define-handler`** macro expands as follows:

```
(define-handler (sensor-blink-msg n)
  (blink n))

==>

(define sensor-blink-msg (lambda (src n)
                           (blink n)))
```

The resulting procedure has an additional parameter `src` that receives the source address of the message. The `src` parameter is not used in this example, but we will see its use later in this chapter.

Note that the message type symbol `sensor-blink-msg` in the call to `bcast` at line (9) refers to the name of the message handler at line 4. SensorScheme dispatches incoming messages to the message handler with the given name. Besides the handler name, macro **`msg`** accepts any number of arguments. The macro constructs a message which is essentially a list of values. The first value in the list is the symbol of the handler name to be called upon reception. Chapter 7 further describes the details of communication in SensorScheme.

The body of message handler `sensor-blink-msg` uses the sensor value in the message to display a number onto the device's LEDs using the `blink` primitive (line 5).

5.4 Extension: moving averages

We continue with an extension to the previous program to explain *closures* and how they benefit programming wireless sensor networks.

Suppose we want to smoothen the sensor readings before broadcasting using an *exponentially weighted moving average.* An exponentially weighted moving average smoothes or averages a sequence of values according to the formula $s_i = \alpha \times v_i + (1-\alpha) \times s_{i-1}$, where v_i are the values obtained from the sensor, s_i are the smoothed values, and α is a 'smoothing factor' between 0 and 1. Since SensorScheme only supports integer values and integer division, we use a numerator-denominator pair to replace α:

$$s_i = \frac{n}{d} \times v_i + (1 - \frac{n}{d}) \times s_{i-1} = \frac{n \times v_i + (d-n) \times s_{i-1}}{d}$$

```
   (define (exp-mov-avg numer denom smooth-val)
 2   (lambda (val)
       (set! smooth-val (/ (+ (* numer val)
 4                             (* (- denom numer) smooth-val))
                          denom))
 6     smooth-val))

 8 (define sensor-avg (exp-mov-avg 9 10 (sense-temp)))

10 (define (time-loop t)
     (call-at-time (+ t 4) time-loop)
12   (bcast (msg sensor-blink-msg (sensor-avg (sense-temp)))))
```

Listing 5.2: exponentially moving average

Listing 5.2 shows the definition of the moving average function and how it is applied in procedure `time-loop` of Listing 5.1.

Closures

Procedure `exp-mov-avg` (line 1 of Listing 5.2) sets up and returns a function that smooths successive values v_i (called `val`) and returns the smoothed values s_i (called `smooth-val`).

Procedure `exp-mov-avg` sets up the smoothing function it returns with the numerator-denominator pair `numer` and `denom`, and an initial value for `smooth-val`. The smoothing function uses variables `numer` and `denom` in the calculation of the smoothed values and `smooth-val` to store the last smoothed value at each calculation, and use it in the next calculation.

The smoothing function at line 2 refers to parameters from the outer procedure: `numer`, `denom` and `smooth-val`. The function is said to be "closed over" its free variables, and is called a *closure*. These closed over variables are accessible to the inner function even when the outer function has terminated, and keep the values assigned to them.

The second definition (line 8) calls `exp-mov-avg` with values for `numer`, `denom` and `smooth-val` and binds the function returned to global variable `sensor-avg`. Every time procedure `time-loop` reads a new sensor value, it calls `sensor-avg` (line 12).

As the example program shows, closures hide computation-local state. In object-oriented programming languages objects perform a similar function, and can be implemented using closures [Kis99, Dic92]. For many of the situations where the functionality of *objects* is required, *closures* are the equivalent for languages containing it.

```
   (ssmodule intruder-single
 2   (require "thesis-base.ss")

 4   (define threshold 15)

 6   (define (>-node l r)
       (if (> (second l) (second r)) l r))
 8
     (define (centroid ls)
10     (let ([sum-v (foldl + 0 (map second ls))])
         (list sum-v
12             (/ (foldl + 0 (map third ls)) sum-v)
               (/ (foldl + 0 (map fourth ls)) sum-v))))
14
     (define-handler (neigh-msg neigh)
16     (set! neigh-ls (cons neigh neigh-ls)))

18   (define neigh-ls ())

20   (define (time-loop t)
       (call-at-time (+ t 16) time-loop)
22     (let* ([v (sense-mag)]
              [me (list id v (* v (x-coord)) (* v (y-coord)))])
24       (when (eq? me (foldl >-node me neigh-ls))
           (send-root (msg neigh-result (centroid (cons me neigh-ls)))))
26       (set! neigh-ls ())
         (when (> v threshold) (bcast (msg neigh-msg me)))))
28
     (time-loop (now))
30   )
```

Listing 5.3: Single hop intruder detection program.

Note that pure functional languages contain a similar notion of closures, but as assignment is not possible for these languages, they cannot be used to capture state in the way shown in this example.

5.5 Intruder detection scenario

After introducing the basic programming techniques and discussing a small program we can now discuss the first application scenario: intruder detection. Listing 5.3 contains a simplified version of the application, receiving data only from direct neighbors. The program has the familiar structure of a single `time-loop` procedure (line 20) that is scheduled to run repeatedly, at 1 second intervals (16 timer ticks).

Within this interval a node may receive messages from neighbors through the `neigh-msg` message handler (line 15). Messages contain a single data item which the program adds to the front of the list bound to the `neigh-ls` global variable (line 16). At every 1 second interval, procedure `time-loop` calls `sense-mag` to obtain a sensor value (line 22) and bind it to variable `v`, and constructs its own message content into variable `me` as a list of the node's network address **`id`**, the sensor reading `v` and the node's coordinates multiplied by the sensor value (line 23). On line 27, the application then broadcasts the message (using primitive `bcast`), if the sensor reading `v` is above `threshold`.

In-between, at lines 24 - 26, calculation of the intruder's position occurs only on the node with the highest sensor reading (line 24). Each node traverses the `neigh-ls` list using the `foldl` function (line 25). When a node has found its own reading to be the highest, it sends the `centroid` of the readings to the network's gateway or *root* using communication primitive `send-root`. The gateway then accepts the `neigh-result` message and further uses the content – the intruder's estimated location.

Reduction

The *higher order* function `foldl` (copied in Listing 5.4 from module `std`, Listing C.2) is used for both calculating the maximum sensor value (Listing 5.3 line 24) and in procedure `centroid` (line 9). The function is an implementation of the *reduction strategy* introduced in Sectionsect-commabstractions. It *folds* or *reduces* a list of values (parameter `ls`) into a single value by repeatedly applying a function (parameter `fn`) on the partial result and the next value in the list. The second parameter, `init`, is the initial value used as the first partial result. `foldl` returns the value of `init` when the list is empty.

The fold operation operates on lists of values independent of the number of items. This method is suitable to sensor network calculations that are based on a variably-sized data set, that may be different from node to node and change over time. The reduction strategy is also at the heart of neighborhood abstractions like Abstract Regions [WM04] and Hood [WSBC04], and macro-programming platforms such as Regiment [NMW07] and MacroLab [SHW08].

```
(define (foldl fn init ls)
    (if (null? ls)
        init
        (foldl fn (fn (car ls) init) (cdr ls))))
```

Listing 5.4: Definition of `foldl`

```
(ssmodule intruder-twohop
  (require "thesis-base.ss")

  (define-const threshold 15)

  ; return node value with maximum reading
  (define (max-node l r)
    (if (> (second l) (second r)) l r))

  (define (process-neighs ls)
    (unless (or (null? ls) (member (caar ls) id-ls))
      (set! id-ls (cons (caar ls) id-ls))
      (max-node-fold (car ls) #f)
      (sum-v-fold (second (car ls)) #f)
      (sum-x-fold (third (car ls)) #f)
      (sum-y-fold (fourth (car ls)) #f)
      (process-neighs (cdr ls))))

  (define-handler (neigh-msg ls)
    ; add direct neightbor's data to neigh-ls
    ; only if node seds its own data
    (when (= src (caar ls))
      (set! neigh-ls (cons (car ls) neigh-ls)))
    ; calculate partial max-node and centroid
    (process-neighs (cdr ls)))

  (define neigh-ls ())
  (define max-node-fold (closure-fold max-node ()))
  (define sum-v-fold (closure-fold + 0))
  (define sum-x-fold (closure-fold + 0))
  (define sum-y-fold (closure-fold + 0))
  (define id-ls ())

  (define (time-loop t)
    (call-at-time (+ t 16) time-loop)
    (let* ([v (sense-mag)]
           [me (list id v (* v (x-coord)) (* v (y-coord)))]
           [sum-v (sum-v-fold 0 #t)]
           [sum-x (sum-x-fold 0 #t)]
           [sum-y (sum-y-fold 0 #t)]
           [msg-ls (if (> v threshold) (cons me neigh-ls) neigh-ls)])
      (set! neigh-ls ()) ; reset list of received neighbors
      (max-node-fold me #f) ; add own data
      (when (eq? me (max-node-fold () #t))
        (send-root (msg neigh-result (list sum-v (/ sum-x sum-v) (/ sum-y sum-v)))))
      (bcast (msg neigh-msg msg-ls))))

  (time-loop (now))

  )
```

Listing 5.5: Two hop intruder detection program.

5.6 Extension: two-hop gossip protocol

Listing 5.5 extends the single hop protocol to use sensor values from the two hop neighborhood. It follows a similar periodic broadcasting method, with the exception that as values from neighbors arrive they are *folded* directly (in procedure `process-neighs`, line 10), to avoid filling the memory with large amounts of neighbor sensor values.

This application uses a procedure called `closure-fold` to create a closure (line 29) that, when called (line 14), folds values and stores intermediate results, or returns the fold result, depending on the second argument. Listing C.1 contains the definition of `closure-fold`.

The application in Listing 5.5 sends a message once every period (line 46). This message contains the node's own sensor value if it exceeded the threshold (line 41), and all values it received from its direct neighbors. When such a message is received by a node, it directly applies the fold operation to all the received values (line 25). Nodes keep track of the neighbors from which they have received and processed values, to make sure no duplicates are used (line 11). Similar to the previous program, all messages from first-hop neighbors are gathered in a linked list (line 23), which is then sent at the next timer interval along with the node's own sensor values.

5.7 Environmental monitoring

```
(ssmodule monitoring
  (require "thesis-base.ss")

  ; definitions of query
  (define period (* 30 16))
  (define duration (* 30 16 60))

  (define (init)
    (if (= (+ (/ (x-coord) 10) (* (/ (y-coord) 10) 4)) 6)
        (list (list id (sense-temp)))
        ()))

  (define (proc l r)
    (append l r))

  ; auxillary functions
  (define (current-epoch)
    (/ (synced-now) period))

  (define recv-fold (closure-fold proc (init)))

  ; handler called when receiveing message from children
```

```
     (define-handler (parent-msg val)
24     (recv-fold val #f))

26   (define (time-loop t)
       (unless (> (+ start-time duration) (synced-now))
28       (call-at-time/synced (+ t period) time-loop))
       (let ([init-val (init)])
30       (call-at-time/synced (+ t (* (- 8 (hops-to-root)) 16))
                          (lambda (t)
32                          (let ([agg-val (recv-fold init-val #t)])
                              (unless (null? agg-val))
34                          (send-parent (msg parent-msg agg-val)))))))

36   (define start-time (synced-now))
     (time-loop (current-epoch))
38
     )
```

Listing 5.6: Environmental monitoring application

Our second application scenario uses a dense sensor network to monitor environmental conditions in a field. It uses the concept of *queries* to obtain particular sensor data, analogous to distributed databases. The application scenario describes two queries, one providing overview information, the other localized full data reporting to detect anomalies.

Listing 5.6 shows the implementation of the environmental monitoring scenario using the *anomaly query* (see Section 3.2.2). The application is structured similar to the intruder detection application in Listing 5.5. It uses a `time-loop` procedure in line 26 to send data at a regular interval (line 34). The implementation makes use of the TinyOS *Collection* protocol described in Section 2.5.1, which sets up and maintains a routing tree. Communication primitive `send-parent` (line 34) sends a message to a node's parent node and `hops-to-root` returns the distance to the root in number of hops. This application furthermore uses a time synchronization protocol ensuring a common time base for all nodes in the network, using the primitives `call-at-time/synced` and `synced-now` to access the network-wide synchronized time. These are analogous to primitives `call-at-time` and `now` described earlier.

Queries

Queries state the sensors to read and operations to perform on the sensor values. As mentioned in the scenario description, the sensing and processing takes place at two separate moments. At the start of each period nodes generate sensor data by calling `init`. Upon reception of sensor data from neighbors they are folded into a single aggregate by calling `proc`.

```
;;; tinySQL pseudo-code for the query:
; SELECT min(moist), max(moist), avg(moist) FROM sensors
; GROUP BY x / 10 + y / 10 * 4
; SAMPLE PERIOD 5m

(define-const period (* 5 60 16))
(define-const duration MAX-DURATION)

(define (init)
  (let ([val (moist-sensor)]
        [area (+ (/ (x-coord) 10) (* (/ (y-coord) 10) 4))])
    (list (list area val val val 1))))

(define (proc l r)
  (group (lambda (l-area r-area)
           (let ([area-val (first l-area)]
                 [min-val (min (second l-area) (second r-area))]
                 [max-val (max (third l-area) (third r-area))]
                 [sum-val (+ (fourth l-area) (fourth r-area))]
                 [count-val (+ (fifth l-area) (+ fifth r-area))])
             (list area-val min-val max-val sum-val count-val))) l r))
```

Listing 5.7: Overview query, producing regional summary information

```
;;; tinySQL pseudo-code for the query:
; SELECT nodeid, moist FROM sensors
; WHERE x / 10 + y / 10 * 4 = 6
; SAMPLE PERIOD 30s
; DURATION 2h

(define-const period (* 30 16))
(define-const duration (* 2 60 60 16))

(define (init)
  (if (= (+ (/ (x-coord) 10) (* (/ (y-coord) 10) 4)) 6)
      (list (list id (moist-sensor)))
      ()))

(define (proc l r)
  (append l r))
```

Listing 5.8: Anomaly query, reporting the localized full data set

The application in Listing 5.6 periodically reads some of its sensors using the `init` procedure (line 29), and constructs a value which it sends to its parent. Upon reception at the parent, the received value is *fold*ed with other values received from children (line 24), again using the `closure-fold` procedure. `closure-fold` applies procedure `proc` repeatedly to the new values arriving and the intermediate result. The sensor value read by `init` is the initial value of the fold. After all values from children have been received, a node sends the folding result to its parent node.

Besides `init` and `proc`, queries should also state the period duration of the sense-and-transmit cycle in variable `period`, and duration of activity of the query with `duration`.

Listings 5.7 and 5.8 show the `init` and `proc` SensorScheme definitions that together form a query for both of the scenario's example queries.

5.8 Extension: dynamic query loading

The implementation presented in Listing 5.6 is capable of executing queries, but still lacks some required properties. First, it only executes a single query at a time. When a different query is needed, the current program one is to be replaced by a new one. Second, while only the definitions of `period`, `duration`, `init` and `proc` change, the entire application is reloaded.

Dynamic evaluation

Listing B.3 shows an improved version, named SSQuery. The application accepts a query in handler `query-msg`, receiving a query ID number `qid`, and the query's `period`, `duration`, `proc` and `init`).

The `proc` and `init` procedures inside the message are transported as linked lists that encode their behavior. Their encoding is equal to the encoding of applications as they are transmitted into the network to reprogram sensor nodes. Chapter 6 describes SensorScheme's program encoding.

When a new query arrives the query functions `proc` and `init` are handed to the SensorScheme interpreter in a call to the `eval` primitive. The interpreter *evaluates* the received data structures representing functions, and returns SensorScheme functions that will be used in calls to `closure-fold` (line 11)

As is common for Scheme implementations, `eval` makes the behavior of the interpreter directly available to user programs. This way, programs can dynamically extend their functionality. Using `eval` to evaluate queries, their behavior and complexity is limited only by the SensorScheme language itself.

When nodes receive a new query, a reference to closure `recv-fold` is added to association list `query-ls`, and the `time-loop` is started (lines 30–31). Multiple queries may be active simultaneously, each operating at a unique period and performing its own sensing and aggregation operations. Nodes receive a query message from the root node using the dissemination protocol used for dynamically loading SensorScheme programs. Chapter 7 explains SensorScheme's method of using of various communication protocols.

Every query sends the aggregated sensor values for the current time period or *epoch* to its parent in a `parent-msg` message (line 28). These messages contain the query ID `qid` and aggregated value `val`. Upon receiving a message from a child node in handler `parent-msg` (line 6) a node uses the query ID to look up closure `handle-parent-msg` and call it with the received value as argument. This directly folds the received value with the intermediate result stored in closure `recv-fold`.

5.9 Logistics

The logistics application scenario (Section 3.3) uses sensor nodes attached to pallets and crates to monitor the transport of a shipment of bananas from the farm to a distribution center. At the banana farm, each sensor node is programmed with a SensorScheme program, called an *itinerary* that tracks the bananas as they move through the logistics process. Listing B.4 contains the itinerary program. The program consists of two processes that will run in parallel.

The first process (line 56) continuously verifies whether the device can find any nearby pallets that contain coffee, by scheduling function `coffee-checker` repeatedly. If found, it signals an alarm by blinking its LEDs and broadcasting a message.

The second process (line 59) is structured as a thread that executes top to bottom, calling procedure `state-loop` several times, one for each stage of the transport process: a) at the farm, b) inside the truck, c) on the harbor dock, and d) inside the shipping container. The procedure continuously checks the conditions at each stage of the transport, and raises an alarm when an error is detected. Only when the condition for transition to the next stage is detected, does the `state-loop` exit. The itinerary then prepares to transition to the next stage by writing the the state change (and time of occurrence) to the on-device log, and setting up the intended destination for the next stage if required.

```
   (define-handler (logistics-msg type kvlist)
 2   (set! msg-ls (cons (list src type kvlist) msg-ls)))

 4 (define properties ())
   (define (set-property! key val)
 6   (set! properties (cons (key . val) properties)))

 8 (define (state-loop alarm? alerter transit?)
     (call/cc
10    (lambda (k)
        (define (time-loop t)
12        (if (transit? msg-ls) (k #t)
              (begin
14              (call-at-time (+ t 80) time-loop)
                (when (alarm? msg-ls) (alerter msg-ls))
16              (set! msg-ls ())
                (bcast (msg logistics-msg 'goods
18                          properties)))))
        (time-loop (now))
20      (exit #f))))
```

Listing 5.9: Definition of procedure `state-loop`

As Listing B.4 shows, the every stage of the itinerary uses a strategy similar to that used in SSQuery: the conditions to check and guard are implemented as SensorScheme functions, given as parameters to `state-loop`. The implementation of `state-loop` is shown in Listing 5.9. It accepts three arguments, `alarm?`, `alerter` and `transit?`, all three functions taking a single argument. Listing 5.9 contains an inner definition of procedure `time-loop` that repeatedly schedules itself, until function `transit?` returns a true value (line 12).

Every period, it sends a `logistics-msg` message (line 17) and receives multiple such message, gathered in `msg-ls`. This list of messages is used to determine transit conditions (function `transit`) or alarm conditions (function `alarm?`).

Other devices broadcast such `logistics-msg` messages. The message contains a *type* – either `goods` for devices on pallets or crates, `transport` for devices inside trucks, shipping containers, or `infrastructure` for access points on the harbor dock and in the distribution center.

The second message parameter is an association list of *properties*. All devices broadcast properties about themselves, such as their destination, owner or location as properties. The tacking device adds or modifies its own properties at the start of the itinerary (line 60) or when changing to the next stage (line 104).

The transportation tracking sensor node on the banana pallet tracks the transport process by filtering the list of received the messages and property lists in them. For example, while on the farm, before loading into a truck, nods check whether they find any other pallets nearby that have a destination different from their own (line 66). If any such messages are received the node will blink its LEDs red (72), to notify personnel. A tracking node will transition to the next state when it has found any `transport` device with a destination of `Rio-harbor` (line 75).

Threads with Continuations

Procedure `state-loop` shows a technique to invoke thread-like behavior in SensorScheme using *continuations*. We will not discuss the nature of continuations in full detail her, but refer the reader to Ferguson and Deugo's description [FD01].

A continuation represents a state of computation. Capturing the *current continuation* at some point in a program evaluation allows one to return to that computational state at any later moment. The `call/cc` primitive called in line 9 of Listing 5.9 is a synonym for the `call-with-current-continuation` standard Scheme procedure. It expects a function of one argument as its sole parameter, which it calls supplying the current continuation as a callable procedure. When the continuation procedure (in argument `k` at line 10) is called (line 12) computation resumes to the point after the call to `call-cc`, which terminates procedure `state-loop`. The procedure called by `call/cc` (line 10) does not return. Instead, it immediately terminates the currently running program with a call to `exit` (line 20).

Communication protocols

Listing 5.10 contains the source code of `send/ack`, a small network protocol used to communicate to harbor and distribution center access points in Listing B.4. Its purpose is to provide a higher reliability single hop communication service, using acknowledgements and retries. The protocol is accessed though a blocking function call, that returns success or failure of the message transmission. Procedure `send/ack` (line 14) takes as parameters the maximum number of retries (`num`), a timeout to wait for an acknowledgement (`timeout`), the message recipient (`dst`), and the message itself (`mess`). The procedure creates a unique message ID (using `make-reqid` on line 15), and uses that to broadcasts a packet `send/ack-msg` (line 23) containing the destination node ID (`dst`), request ID (`reqid`) and the message itself (`mess`).

Upon receipt of this message, the `send/ack-msg` message handler is invoked (line 3). When the intended recipient receives the message, it returns a message

```
(define exit (call/cc (lambda (k) k)))

(define-handler (send/ack-msg dst reqid mess)
  (when (eq? dst id)
    (bcast (msg send/ack-ack src reqid))
    (handle src mess)))

(define-handler (send/ack-ack dst reqid)
  (when (eq? dst id)
    ((cdr (assoc/remove! reqid waiting-reqs)) #t)))  ; return from continuation

(define waiting-reqs ())

(define (send/ack num timeout dst mess)
  (let ([reqid (make-reqid)])
    (define (ack-not-recvd t)
      (lambda (t)
        ((cdr (assoc/remove! reqid waiting-reqs)) ; call continuation
                   (if (> num 0)                  ; if still more retries
                       (send/ack (- num 1) timeout dst mess)
                       #f))))            ; return #f when all attempts tried

    (bcast (msg send/ack-msg dst reqid mess))
    (call/cc (lambda (k)                               ; continuation in k
               (assoc-put! reqid k waiting-reqs)       ; store continuation
               (call-at-time (+ (now) timeout)
                             ack-not-recvd)
               (exit #t)))))
```

Listing 5.10: implementation of an acknowledged send operation using `call/cc`.

to the sender (line 5). Every message handler implicitly declares a parameter `src` which is bound to the node ID of the node sending the message.

The `send/ack` protocol acts according to the layered communication stack model: when message arrives on the node, the reliable communication protocol passes the message content `mess` to a higher layer message handler through a call to `handle` (line 6). See Listing C.1 for the definition of `handle`.

After the original sender sends the message, it creates a continuation (line 24) and stores it in association list `waiting-reqs` with the request ID as key (line 25). It then sets up a timer function that will be called after `timeout` timer ticks have passed, and immediately terminates the currently executing task by calling `exit` (line 28).

When an acknowledgement message arrives at the original sender through handler `send/ack-ack` (line 8), the continuation is removed from `waiting-reqs`, and invokes it returning true, returning from `send/ack` to its original caller (line 10).

At timer expiry procedure `ack-not-recvd` is called (line 27), which retrieves the continuation and invokes it with the return value of the next attempt of sending the same message (line 20). If no more retries are left, the continuation is called with return value false (line 21).

Note that `send/ack` behaves as a blocking call – returning only after an acknowledgment has been received or all timeouts have passed. While the call is blocking, other events may execute, however. The task that started its execution terminates when `exit` is called. Procedure `send/ack` returns in the task context of the acknowledgement arrival or timer event.

Other uses of continuations

Besides the programmatic use shown in this example, the SensorScheme interpreter uses continuations in a number of contexts. First, when a program calls `bcast` to send message, the interpreter retrieves the current continuation before it sends the message. After transmission has completed (when the `sendDone` event is signaled) the saved continuation is invoked to resume the computation at the point of returning from the call to `bcast`.

Similarly, reading sensors is a split-phase operation in WSN operating systems. First, the read operation is started, and when the sensor value is available from the A/D converter and event is signaled. SensorScheme uses continuations to implement a blocking sensor reading primitive.

5.10 Conclusion

This chapter has shown by way of a number of example programs the programming techniques available with SensorScheme. As a descendent of the Scheme language, SensorScheme provides features and mechanisms not found in other languages for wireless sensor network platforms, such as closures, higher-order functions, continuations and dynamic evaluation, and a language-integrated communication mechanism.

We have shown how these techniques can be put to good use to write concise WSN programs that use a number of techniques used elsewhere in the WSN research field: the reduction strategy to aggregate sensed values from multiple nodes into a single summary result, and the use of blocking I/O operations to simplify the program's flow of control. Additionally, SensorScheme contains properties unique to WSN platforms, such as the `eval` procedure, which enables dynamically loading and executing additional program functionality.

SensorScheme's communication mechanism simplifies writing communication protocols as the application programmer is not burdened with the placement and encoding of the message content. Instead, communication proceeds as a remote procedure call, requiring minimal amounts of code.

SensorScheme is a multi-paradigm programing language, providing functional programming concepts as well as imperative and object-oriented. The example programs in this chapter have shown its merit as a multi-paradigm language. One may use its functional nature where this is most applicable – as in the use of the reduction strategy and aggregation in queries, and imperative nature where this is most practical - to interact with sensors and actuators, to use scheduled execution (through `call-at-time`) and for communication, and – essentially, to dynamically load and modify entire applications or partial functionality, as the SSQuery application has shown.

This chapter has shown how to use the discussed language features and techniques to implement the intruder detection (Section 3.1), environmental monitoring (Section 3.2) and logistics (Section 3.3) application scenario's.

The next chapters will describe the SensorScheme language and platform more formally (Chapter 6), as well as its communication mechanism in Chapter 7. Subsequenly, Chapter 8 describes an extension to the platform described before, and shows its use in an implementation of the smart office application scenario. We finish this work with performance evaluations of the SensorScheme programs discussed in this scenario.

Chapter 6

SensorScheme design

This chapter describes the SensorScheme language and interpreter. It describes the language used in chapter 5 to implement the application scenarios and describes the reference implementation for an interpreter to execute these programs in low power sensor nodes. The SensorScheme reference implementation is developed as part of the research described in this dissertation. It is available on-line at `www.sensorscheme.net` or as part of the TinyOS 2 source distribution [Sou].

6.1 Platform overview

We will start this chapter by giving a coarse overview of the operation of the SensorScheme platform, and refine de descriptions of the constituent parts in the next sections.

Figure 6.1 shows SensorScheme's overall operation showing the transformation of SensorScheme module and library files on the top into a binary code image or network message containing a SensorScheme program on the bottom. These transformations all take place on a regular PC-class device, which stores and processes files to produce binary representations of programs that execute on WSN nodes, shown on the bottom of the figure.

The SensorScheme platform consists of the source and intermediate files (the 'sheets' in Figure 6.1) and processes (the 'wavy blocks') that transform the contents of source files into other files. Some collections of files are organized into libraries (the 'stacked sheets').

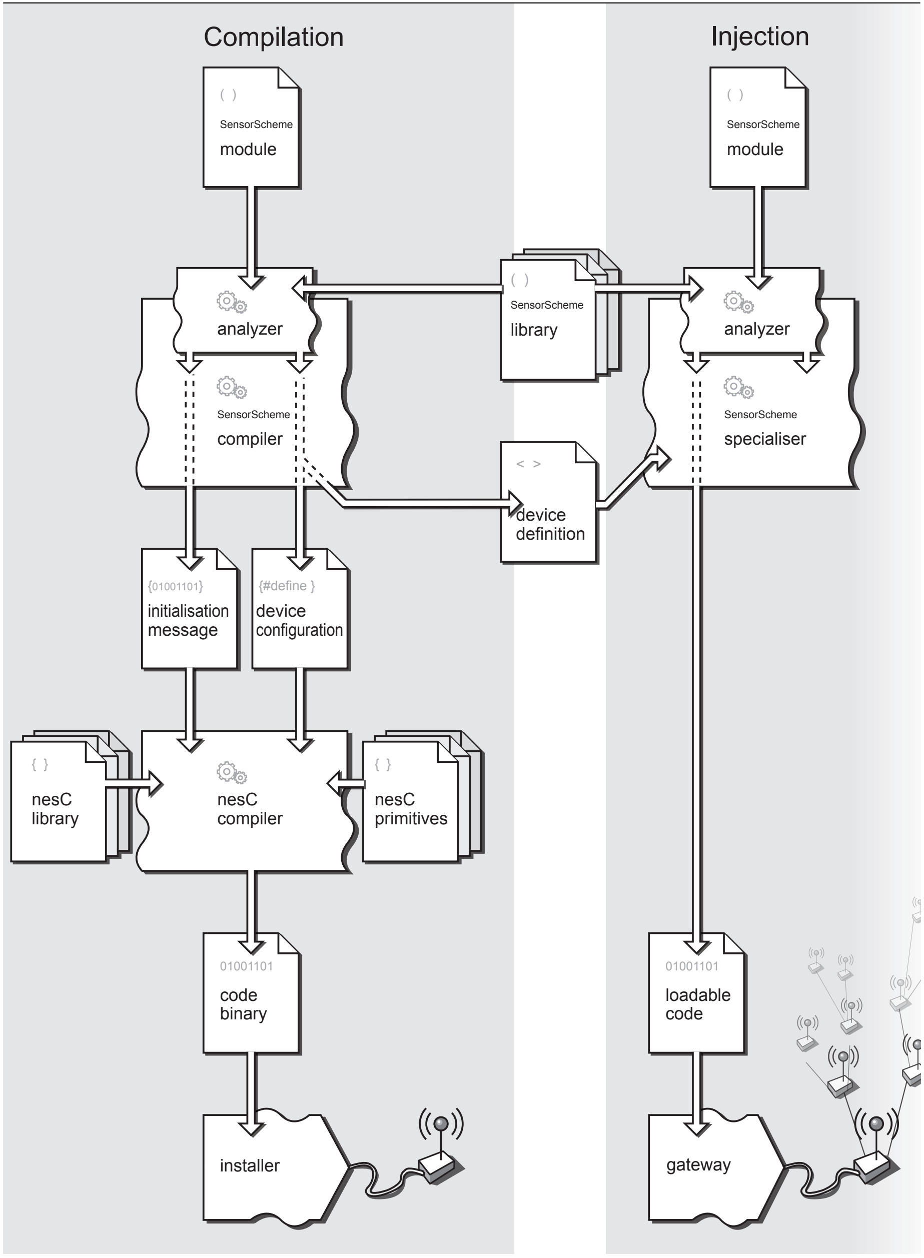


Figure 6.1: SensorScheme compilation and injection phases

The source and intermediate files may contain a variety of content as indicated by a signature in their top left corner as follows:

- **( )** : SensorScheme source files. These files are all structured as *modules* (see Section 6.2) and may contain either a program or a library.
- **{ }** : NesC source files. These files are all part of either the nesC Library part of the TinyOS platform or a library of Sensorscheme primitives written in nesC.
- $< >$: General configuration files containing data to be transferred between processes.
- **01001101** : binary files, containing programs in a machine-readable or executable form.

Arrows connect the source files and processes, showing the flow of information from the top of the graph – the source – to the bottom – the target. All processes accept input files through inward arrows, and produce files through outward arrows. We start our description on the bottom part of the graph, the target of SensorScheme, and proceed to the top, that starts with input of SensorScheme source files.

6.1.1 Phases

The use of SensorScheme consists of two phases, called ‘compilation’ (left part of Figure 6.1 and ‘injection’ (right side of Figure 6.1). Compilation starts with a SensorScheme source module file and produces a code binary to be installed into sensor nodes’ code memory. Injection also starts with a SensorScheme source module and produces an *injection* message containing binary code to wirelessly transfer to sensor nodes in a network. Before these nodes can successfully receive an injection message these nodes must contain a binary code image containing basic SensorScheme functionality produced with the compilation phase.

6.1.2 Compilation

To use the platform, a network of WSN nodes must be loaded with a binary code image containing the SensorScheme interpreter. The compilation phase builds a *code binary* which the *installer* installs on a connected node, as the bottom left of Figure 6.1 shows.

The SensorScheme interpreter is written as a TinyOS application and compiled using the nesC compiler. The SensorScheme interpreter is configurable. The compilation phase configures the interpreter with a set of primitive procedures, or *primitives*. A *device configuration* file – a nesC source file containing macro definitions – configures the set of primitives included in the interpreter. Primitives are procedures available to SensorScheme programs to interact with the operating system and peripheral devices, and perform operations on basic data types. Primitives are written in nesC. The *nesC compiler* compiles the configured set of primitives from the *primitives library* along with the nesC library into the *code binary* containing the interpreter implementation.

Additionally, configuration may specify a SensorScheme source program to execute upon initialization of the nodes as an *initialization message*. SensorScheme's interpreter configuration is described in Section 9.2 as part of the description of modules.

6.1.3 Injection

With a configured interpreter installed by the *compilation phase* a SensorScheme sensor network may be deployed in its location of use, and start the *injection phase*. A network is in contact with a *gateway*, a PC-class computer, as shown on the bottom right of Figure 6.1. Subsequently, *loadable code* may be injected into the network from the gateway. The injection phase transforms a *SensorScheme source* program into a message containing the *loadable code* that is transported to each of the nodes in the network. Upon reception, each node executes the code in the message.

The program executed upon initialization described above takes the shape of an *initialization message* stored in a node's code ROM. After a node boots up it reads the *initialization message* and executes the code in the message, identical to the execution of a message containing *loadable code* in the injection phase.

6.1.4 Compiler, specializer and analyzer

Both the compilation phase and injection phase take a *SensorScheme source* module as input, at the top of the figure. These source files contain the application or interpreter configuration to be installed and executed on sensor nodes.

The *analyzer* reads and analyzes the source module and modules from the SensorScheme *library* it refers to. It distills from these a set of definitions, primitives and top level expressions that must be present on a sensor node to execute the program in the source module. All of these for the input to the next, lower stages of processing in the following ways:

The definitions and top level expression in the module are combined into an *initialization message* during the compilation phase, or into an *injection message* during the injection phase. The precise operation of the analyzer process is detailed further in Section 6.2.1 and the definition of *modules* in Section 6.2.

The required primitives distilled from the source modules are used to build the device configuration: only those primitives that are required by the source module at the start of the compilation phase will be included into the code binary installed on a WSN node.

Section 6.2 discusses more extensively the content of SensorScheme module files, the detailed operation of the analyzer and and how modules are to be used to compose Sensorscheme applications.

6.1.5 Loading and execution

Now that we have discussed the operation of the SensorScheme platform, we will focus on how the source is loaded and executed on sensor nodes. SensorScheme's compilation phase compiles the application alongside the TinyOS libraries into a single binary which is then written into each device's program memory.

The following shell commands build a TinyOS application from Sensor-Scheme source module `sensor-blink` and compile and install it on a device:

```
# First build a TinyOS application from SensorScheme program:
$ ssbuild sensor-blink.ss
--> Generating NesC application for module "sensor-blink.ss" ...
    Target directory is /SensorScheme/scripts/
    Module "sensor-blink.ss" compiled to size of 41 bytes
    Code occupies 39 cells of memory in node
    copying additional build files...

# build and install application with TinyOS tool chain
$ make telosb install.1
--> compiling src/SensorSchemeAppC to a telosb binary
    compiled src/SensorSchemeAppC to build/telosb/main.exe
            34526 bytes in ROM
             1384 bytes in RAM
    writing TOS image
```

Second, the injection phase wirelessly loads programs into a running sensor network. All nodes in the network need to run a binary containing a properly configured SensorScheme interpreter, as generated by the compilation phase procedure described above. A network running this binary can then be used to 'inject' new applications into:

```
$ ssbuild thesis-base.ss
...
$ make telosb install.1
...
$ ssinject  sensor-blink.ss
--> injecting /SensorScheme/scripts/sensor-blink into network...
    done.
```

6.2 Modules

All SensorScheme source code is contained within a module. Modules contain the source for applications, device definitions as well as library code. Modules contain module clauses, definitions and expressions. A module may refer to, or *require*, other modules, which allows expressions in the referring module to refer to *provided* definitions in the required *module*.

Library modules *provide* a set of definitions for use in applications. Listing C.1 and C.2 show two examples of library modules, used by the example applications in Chapter 5. A special use of library modules is their use as *base module*, containing the definition of a SensorScheme interpreter configuration. Such a base module the input of the compilation phase, resulting in a configured interpreter, compiled and installed sensor nodes s code binaries.

The use of a module system is not defined in the R^5RS Scheme standard. Several Scheme implementations do define an implementation-specific module system. SensorScheme's module system is similar to and inspired by that of PLT Scheme [FS09].

Modules use Scheme syntax already introduced in Chapter 5. Below we describe the module contents using a pseudo-code syntax we will be using throughout this chapter. A fixed-width font denotes literal source code, with SensorScheme keywords in bold. Words in variable-width italics font denote variables – *var*. Lists, between brackets `(` and `)` may contain a range of values bound to subscripted variables var_1 ... var_n, where the range is possibly empty.

Module file

```
(ssmodule name

  elem_1 ... elem_n
)
```

A SensorScheme module file contains a module definition. It includes the module name *module-name* and a module body as a list of module body elements, shown as $elem_1$ ... $elem_n$ in the above definition.

Each of the module body elements may be one of the following syntactical forms:

Require clause

```
(require (lib name_1) ... (lib name_n))
(require path_1 ... path_n)
```

A *require clause* states the library name or file path of another module required by the current module. The named module is found and analyzed, and all its *provided* definitions are made available to expressions in the current module.

Provide clause

```
(provide name_1 ... name_n)
```

A *provide clause* names a list of definitions provided by the current module for inclusion into other modules using a require clause. The provided definitions may be `define` expressions as well as macro definitions described below, and may be defined directly in the current module (possibly after the provide clause) as well as included from other modules through a *require clause.*

Include clause

```
(include name_1 ... name_n)
```

An *include clause* acts as a use of the named definitions. It has the effect that the named definitions will be included into the set of referred definitions, similar to the appearance of the name of the definition in initialization expressions or definitions according to the procedure described in Section 6.2.1. The include clause is used to include a definition into a program when there is no reference to it from any part of the program, particularly for the inclusion of message handlers.

Macro definition

```
(define-macro name macro-expr)
```

A *macro definition* defines a macro. Macros transform clauses or expressions in a module file into other, more simple ones. The *macro-expr* expresses the transformation of clauses or expressions named by *name*. Section 6.3 describes SensorScheme's macro-expansion process. Note that macro-expansion operates on module clauses. Module clauses following macro-definition *name* in the same module may contain macro-expressions *name*, which will be expanded according to expansion *macro-expr*. Macro definitions may be exported from the defining module using a provide clause.

Primitive definition

```
(define-primitive name (kind component—name))
```

A primitive definition defines a primitive procedure. Primitive definitions consists of a name, kind and the nesC component name in which the primitive is implemented. Several different kinds of primitives are defined: `simple`, `sender` and `receiver`, as described in Section 6.6. Primitive definitions may be exported from the defining module using a provide clause.

Expression

```
(define name expr)
expr
```

Module clauses any other than those defined above are regarded as expressions, to be *evaluated* in an interpreter for sensor nodes. Section 6.5 expresses the syntax and semantics of SensorScheme expressions and the operation of the interpreter. Expressions are either definitions, or *initialization expressions*.

6.2.1 Analyzer

Both the compilation and injection phase in figure 6.1 use the *analyzer* process to read the SensorScheme module file at the start of each phase. The *analyzer* process reads a single module and all modules referred to in its `require` clauses, to produce a list of *initialization expressions* and a set of variable and primitive definitions required to evaluate the initialization expressions.

The process of analyzing a module involves a number of steps.

Read and look up modules. Read the source file input to either phase. Source files contain a single module definition as defined in section 6.2. When `require` clauses are encountered in the module the referred file is looked up and its contents read and processed according to the same procedure.

Expand macros. SensorScheme uses macro's to extend the syntax of programs. All expressions read are macro-expanded at this point. Section 6.3 describes SensorScheme's macro expansion method.

Find used definitions. The source program and required modules contain definition and expression clauses describing the program to execute on sensor nodes. Not all definitions, however, are required to be loaded onto nodes. As an example, in case a library module containing a collection of general purpose procedures is required, only those that are actually referred to in a program need to be present in the SensorScheme interpreter. All others should not be loaded to preserve memory.

The analyzer finds all definitions used by a program with the following process: All expression clauses that are not **`define`** expressions in the source module and all required modules are called *initialization expressions*, and will be evaluated at the start of the program. The analyzer subsequently processes these expressions to find references to variables in the global environment. The definitions of these referred variables are themselves checked for occurrences of global variable references, and so on, recursively, until all global variable references have been identified. The result is a set of **`define`** expressions and **`define-primitive`** clauses required on sensor nodes to execute the source program. The set of primitive definitions is used to configure the interpreter in the compilation phase, while the global variable definitions will be packed in an injection message in the next step.

Construct Injection message. To run a program on sensor nodes, the definitions of all referenced global variables along with the initialization expressions need to be transferred to a sensor node. A message containing executable expressions is called an *injection message*. Upon reception at a sensor node the SensorScheme interpreter evaluates the message content – a list of expressions – which starts the program's execution. Section 6.5 describes the evaluation method used by the SensorScheme interpreter.

An injection message is a regular message. Messages may contain multiple SensorScheme values of arbitrary complexity and size. As a consequence of

SensorsSensorSchemecheme's homo-iconic nature, expressions are SensorScheme values, which may be sent using SensorScheme's communication facilities, described in Chapter 7. The compilation and injection phases in Figure 6.1 will use the injection message to deliver the program to sensor nodes.

6.2.2 Typical use

This section has described the two-phase approach to creating and using a SensorScheme interpreter on sensor nodes, without shedding much light on the actual use of these SensorScheme tools.

A common use case for SensorScheme-enabled sensor network deployments is to create a *platform base library*, similar to the `thesis-base.ss` library used throughout this work, and shown in Listing C.1. The library `includes` and `provides` primitives needed for general program execution, such as type predicates, arithmetic operations and timer control, and primitives to access the hardware peripherals and communication protocols available on the devices. Base libraries furthermore `require` and `provide` the standard library `std.ss` (shown in Listing C.2) and optionally other libraries.

When deploying a sensor network, a SensorScheme interpreter is compiled and installed on all nodes of the network that uses the base library as the source program input for the compilation phase. This interpreter contains all the primitives `included` in the base library, but does not start any program when the sensor nodes start up or reset.

Programs written for this platform, that need to be loaded onto the nodes must `require` the base library in their source module, as is the case of the sample implementations of the scenarios, shown in Appendix B. The injection phase wirelessly transfers the programs to the deployed sensor nodes.

In some cases, (for example for the environmental monitoring scenario) a program should be loaded in the nodes directly when they boot up. This is arranged by compiling an interpreter not from the base library, but to use the program to load as source program input of the compilation phase. During program development this may be desirable as well, to avoid the need to wirelessly load a source program after every change. Similarly, when developing using the TOSSIM sensor network simulator for TinyOS, our experience has been to prefer compiling in the program under development, rather than load the program using the simulated network when the simulation is running.

6.3 Macros

Just like other Lisp and Scheme implementations, SensorScheme contains a macro facility to extend the syntax of the language. SensorScheme's macro system is adopted unmodified from R^5RS Scheme [ADH+98] – using the `syntax-rules` pattern matching syntax, as well as the `syntax-case` pattern definition syntax [DHB93] proposed by Dybvig. We will not describe their operation in this work, but instead refer to their respective definitions.

Macro's transform a Scheme expression starting with a symbol which is the name of a defined macro into another expression, according to the transformation defined in the macro definition. Expressions whose (symbol) name is not defined as a macro are not transformed. Next, macro-transformation transforms sub-expressions of the module-level expressions and clauses and transforms each expression that is defined as a macro. Chapter 5 shows some examples of macros and the expressions they transform into.

The process of transforming an expression is called *macro expansion*. It takes place as part of reading the source modules and required modules as part of the analyzing process. The analyzer reads clauses in a module one by one and performs macro-expansion on each clause, and interprets the clause resulting from the transformation according to the rules defined in Section 6.2.

SensorScheme recognizes only a small number of module clauses (see Section 6.2) and primitive expressions (see Section 6.5). All other special forms or expressions (that are not procedure calls) are implemented as macros that transform into the recognized set of clauses and primitive expressions. For example, SensorScheme uses macros to implement a number of standard Scheme special forms such as **`let`**, `cond`, `and` and `or`, and the **`define`** procedure definition syntax. Additionally, SensorScheme defines a set of syntax extensions for communication and peripheral access such as **`define-handler`**, **`msg`** and **`define-event`**. Listing C.3 shows the macros defined in the SensorScheme standard library.

6.4 Representations and data types

SensorScheme uses three distinct *representations* for values and expressions.

SensorScheme module files contain a textual notation of values and expressions – the *external representation*.

Programs and data reside as *values* in the memory of sensor nodes – the *internal representation*. SensorScheme values are allocated from a pool of equally-sized *cells*, as proposed in Chapter 4.

Additionally, sensorScheme employs a *network representation*, to transfer programs and data from the gateway into the network, as well as during communication between sensor nodes. Chapter 7 describes SensorScheme's communication method and the network representation.

The three representations have separate domains. The external representation is used only in SensorScheme source files, while the internal representation only resides in sensor nodes' memory, and the network representation exists only during communication. There is a direct correspondence, however, between the representations. Descriptions of SensorScheme values in this dissertation will always use the external representation, even when the values reside only in a sensor node's memory.

This section describes the data types all SensorScheme values are comprised of. SensorScheme uses a subset of the data types defined by the Scheme R^{5}RS standard: the data types *numbers* (fixed bit width integers only), *symbols*, and *pairs*, and singleton values (*true* and *false*) and the *null* value. All values belong to one of these types. Procedures and primitives are represented as expressions (see Section 6.5), and are not distinct data types, but created from these basic types.

The data types vectors, characters, strings, ports and arbitrary precision numbers (including floating point), also present in R^{5}RS Scheme are not part of SensorScheme. We have limited the set of valid data types for SensorScheme to ensure their efficient storage in memory. The Scheme data types omitted from SensorScheme can either be emulated using the provided types (such as vectors, which are similar to lists constructed from a chain of pairs) or do not play an important enough role in WSN applications to warrant their inclusion. (WSN platforms do not contain display devices to show strings and characters; no file systems are present to perform file I/O using ports.)

Below follows a description of each of the SensorScheme data types, and their external and internal representations. SensorScheme adopts Scheme's external representations, limited to the subset of data types supported. The description of each data type also refers to the primitive procedures that manipulate values of the data type, and the type predicate to test for membership of that type.

Numbers

Numbers are simple primitive types that contain fixed bit width integer numbers. The current implementation uses signed integer numbers of 31 bits wide to ensure a space-efficient storage, as described further in Section 6.7.1. As is common in most programming languages numbers are used in arithmetic operations, comparisons and bitwise operations.

Numbers are also used as unique identifiers or network addresses for sensor nodes. The Scheme notations for integer numerical constants also apply for SensorScheme numbers. Arithmetic operations `+`, `-`, `*`, `/`, `%`, operate on and produce numbers. Comparison operators `<`, `<=`, `=`, `>`, `>=` compare numbers and produce a boolean value. Membership to the *number* type can be tested with the `number?` predicate procedure.

The external representation of numbers consist of a sequence of decimal digits `0` – `9` optionally preceded by a minus sign `-`[1].

Symbols

Like any programming language, SensorScheme uses *identifiers* in its program texts, called it symbols in Lisp-like languages. As we have introduced already in Chapter 5, symbols are also present at run-time for a number of uses. In the external representation symbols have a textual form.

Symbols are written as a sequence of any of the following characters: alphanumeric characters `A` – `Z`, `a` – `z`, `0` – `9`, plus the special characters `+`, `-`, `!`, `$`, `%`, `&`, `*`, `/`, `:`, `<`, `=`, `>`, `?`, `_`, `^`, `~`. Symbols may not start with a digit.

In the internal representation, available at runtime, symbols are represented as unique numeric values. Symbols with identical external representation have identical numeric value, and symbols with different external representation have different numeric values. Test for equality with the `eq?` procedure is the only operation permitted on symbols. The type's membership procedure is `symbol?`.

The relation between the external representation and the internal, numeric representation is network-wide. This relation is maintained by the gateway computer, and updated every time new programs or data are sent into the network. This network-wide consistency means that symbols on one device may be referred to by another device by transporting the symbol reference across the network. This way symbols play a crucial role within Sensorscheme sensor networks.

Booleans

The boolean values *true* and *false* (written **`#t`** resp. **`#f`**) are the sole two values of this type. They are the result of comparison operations and used as the choice value in conditional expressions. The boolean type predicate is `boolean?`.

[1]Binary, octal and hexadecimal notation are also supported using the standard Scheme prefixes, but we do not make use of these in this document, and leave them out for simplicity

Booleans are singleton values, only a single instance of each is available in a node's memory. The internal representation of singletons is a unique value that is not part of any of the other data types.

Null

The null value is another special singleton value. The null value is commonly used to mark the end of a list. It is written `()`, and `null?` tests for the null value.

Pairs

Pairs are the sole constructive data structures within SensorScheme. A pair contains two values. The values in a pair may be of any of the available data types. The operations permitted on pairs are *construction* (the `cons` procedure), *reference* (the `car` and `cdr` procedures) and *modification* (the `set-car!` and `set-cdr!` procedures). Values of type *pair* within other pairs represent references to the storage location of the contained pairs. Type membership is tested using the `pair?` predicate.

The external representation of pairs is a special case of the list notation. *Lists* are data structures constructed from multiple pairs. A list is a sequence of pairs where the `car`'s of all pairs contain the values in the list and the `cdr`'s refer to the next pair in the list, or to the empty list `()` as the list's end. Lists are written as the external representation of the list contents separated by whitespace, and all between brackets `(` and `)`. Instead of parentheses `(` and `)` square brackets `[` and `]` may be used.

The last item of the list may be written preceded by a `.` surrounded by whitespace. This *dot* notation is used primarily in case the last item of a list is not a null value `()`. Using the dot notation a single pair constructed by evaluating `(cons a b)` is written `(a . b)`.

The same value may be written in several different ways, each translating into the same in-memory data structure. The value `(1 2 3)` translates to the same internal representation as `(1 2 3 . ())` or `(1 . (2 . (3 . ())))`.

Other notations

Besides the external representations of data types described above, the external representation may use some other notations as well.

The quote shortcut notation `'`v translates to an internal representation that is equal to the notation (**`quote`** v) for any value v.

Whitespace separating values may consist of common whitespace characters space, tab and newline. Additionally, comments may take the place of whitespace. A comment starts with a semicolon ; and continues until a newline character. Whitespace and comments are present only in external representation and are not part of the internal representation.

6.5 Expressions

Expressions, contained in SensorScheme modules (see Section 6.2) describe the program to execute on the SensorScheme interpreter on sensor nodes. This section describes the semantics of executing these expressions inside the interpreter.

We can express the execution semantics of the interpreter with an implementation of `eval`. Appendix A presents a definition of `eval` as a SensorScheme procedure. `eval` is a *meta-circular interpreter* – an implementation of an interpreter expressed in the language it interprets. The SensorScheme interpreter executing on sensor nodes is itself written in nesC, and displays behavior identical to that of the `eval` function listed in Appendix A.

The remainder of this section discusses the operation of `eval`. We use a imperative pseudo-code notation with pattern matching on SensorScheme values to stress the separation between the SensorScheme language and the interpreter implementation. We will explain the various notational conventions on first use.

For notational convenience we denote `eval` as **E**. Additionally, the evaluation process makes use of function **A**, equivalent to procedure `apply` in Appendix A.

Function **E** accepts as arguments an expression *expr* to evaluate, and a *local environment* ε. The expression to evaluate – *expr* – is a SensorScheme value. Its type or structure expresses the computation to perform. The expression to evaluate may take the shape of one of the seven different *primitive expressions* according to the definition of **E** below. All other expressions can be defined in terms of primitive expressions.

The result of evaluating an expression is a SensorScheme value. The process of evaluation may also cause side-effects to occur. The side effects performed by the interpreter are definition and assignment of variables, and the operations performed by primitive procedures.

The local environment ε holds bindings to variables that are accessible to expression *expr* and its sub-expressions. Initially the local environment is empty. Besides the local environment ε function **E** accesses the global environment **G** holding variables that are accessible from any location in the program.

The definition of function **E** contains operators that bind or retrieve variables from either environment. Section 6.5.2 further describes these operations of the environments.

6.5.1 Primitive expressions

The SensorScheme evaluator evaluates an expression by examining the data type and structure of the expression argument as a SensorScheme value. Expressions may consist of combinations of the following primitive expressions. Our notation contains patterns on argument *expr* of function **E**. When the value of *expr* matches the pattern, pattern variables (in *slanted* font) are bound to the subexpressions of *expr* and the corresponding function part is evaluated. Otherwise pattern matching proceeds to the next pattern of function **E**.

The implementation of functions **E** and **A** in Appendix A performs the pattern matches using conditional expressions on type predicates and combinations of `car` and `cdr`.

Literals

```
E(expr, ε) when (number? expr)  |
                (boolean? expr) |
                (null? expr)
   return expr

E((quote literal), ε)
   return literal
```

When the expression *expr* is a literal value – number, boolean, or the empty list (null) – **E** returns the expression *expr*. When **E** evaluates a quote expression – a list with the first element the symbol `quote` and the second element a literal value – it returns the literal value within the quote expression.

Note that we match the type of argument *expr* in the 'when' clause using the type predicate primitives described in Section 6.4.

Variable references

```
E(expr, ε) when (symbol? expr)
   return ε[expr]   if  expr ∈ ε
          G[expr]   if  expr ∈ G
          error     otherwise
```

Any expression that is a symbol refers to a variable. If a variable binding for *expr* is present in the local environment ε **E** returns the value bound to *expr* in environment ε. Otherwise if a binding for *expr* exists in the global environment, the binding for *expr* in **G**. It is an error to refer to a variable that is not bound in either the local or global environment. See Section 6.5.2 for the description of operations on environments.

Conditionals

```
$\mathbf{E}(\texttt{(if}\ pred\ conseq\ alt\texttt{)},\ \varepsilon)$
    return $\mathbf{E}(alt\ ,\ \varepsilon)$       if $\mathbf{E}(pred,\ \varepsilon)\ =\ \texttt{\#f}$
           $\mathbf{E}(conseq\ ,\ \varepsilon)$ otherwise

$\mathbf{E}(\texttt{(if}\ pred\ conseq\texttt{)},\ \varepsilon)$
    return #f                 if $\mathbf{E}(pred,\ \varepsilon)\ =\ \texttt{\#f}$
           $\mathbf{E}(conseq\ ,\ \varepsilon)$ otherwise
```

A conditional expression tests the predicate expression *pred* and, depending on the result of *pred*, returns the value returned by evaluating either expression *conseq* or *alt*. **E** recursively invokes itself to evaluate the sub-expressions *pred*, *conseq* and *alt*. Each invocation of **E** may cause side effects, therefore the order of evaluation the sub-expressions is relevant. A conditional expression first evaluates *pred*. If the result is any value other than `#f` *conseq* is evaluated and its result returned. Otherwise *alt* is evaluated and its result returned, or `#f` is returned if *alt* is not present.

Definitions

```
$\mathbf{E}(\texttt{(define}\ var\ value\texttt{)},\ \varepsilon)$
    $\mathbf{G}[var] \leftarrow \mathbf{E}(value,\ \varepsilon)$                  if $var \in \mathbf{G}$
    $\mathbf{G} \leftarrow \mathbf{G} \cup \{var \mapsto \mathbf{E}(value,\ \varepsilon)\}$ otherwise ;

    return ()
```

A definition extends or updates the global environment with a binding of variable *var* to the result of evaluating expression *value*. When variable *var* is present in the global environment, the value bound to *var* in **G** updated with the result of evaluating *value*. If not present, the definition extends **G** with a binding of variable *var* to the result of evaluating *value*. A definition always returns `()`[2].

[2]This is a deviation from the Scheme standard where the result of definition or assignment is an unspecified value

Definitions show the dual paradigm nature of SensorScheme: The first body expression performs a side-effect modifying environment **G**. The next expression, separated by a semicolon, produces the return value of the definition expression.

Assignments

```
$\mathbf{E}$((set! $var$ $value$), $\varepsilon$)
    $\varepsilon[var] \leftarrow \mathbf{E}(value,\ \varepsilon)$  if $var \in \varepsilon$
    $\mathbf{G}[var] \leftarrow \mathbf{E}(value,\ \varepsilon)$  if $var \in \mathbf{G}$
    error otherwise ;

    return ()
```

An assignment expression replaces the value bound to variable *var*with the result of evaluating expression *value*. It is an error to assign to a variable that is not bound in either the local or global environment. Similar to a definition expression, assignment performs a side-effect. An assignment expression always returns the empty list `()`.

Lambda expressions

```
$\mathbf{E}$((lambda ($var_1$ ... $var_n$) $expr_1$ ... $expr_m$), $\varepsilon$)
    return (proc $\varepsilon$ ($var_1$ ... $var_n$) $expr_1$ ... $expr_m$)
```

A lambda expression defines a procedure. It consists of a list of formal parameters (var_1 ... var_n) followed by the function body as one or more body expressions $expr_1$ ... $expr_m$.

A lambda expression returns a *procedure*, a special expression, named `proc`, that captures the local environment ε in effect at the moment of evaluating the lambda expression. A *procedure call* that calls it some later point makes the variables in the captured environment available to the expressions in the procedure body.

Procedure calls

```
$\mathbf{E}$(($fn$ $arg_1$ ... $arg_p$), $\varepsilon$)
    return $\mathbf{A}(\mathbf{E}(fn,\ \varepsilon)\ \mathbf{E}(arg_1,\ \varepsilon)\ \ldots\ \mathbf{E}(arg_p,\ \varepsilon))$
```

Any expression that is not one of the above primitive expressions is a procedure call. The expression fn evaluates to a primitive or procedure value. The zero or more argument expressions arg_1 ... arg_p all evaluate to an argument value.

Expression fn may be any expression and should evaluate to a *procedure* as produced by evaluation of a lambda expression or a *primitive*. Usually fn is either a symbol, referring to a bound procedure or primitive value, or a lambda expression, which is then evaluated to a procedure. Naturally, more complex computations to obtain a procedure are possible.

As a consequence of possible side-effects during the evaluations of fn and $arg_1 \ldots arg_p$ (such as reassignments to variables in ε) their order of evaluation is relevant to the semantics of the program. The interpreter evaluates the procedure expression fn and each of the arguments *exprs* in left to right order[3].

Apply

```
$\mathbf{A}(prim,\ arg_1 \ldots arg_n)$ when (primitive? $prim$)
    return $prim(arg_1, \ldots, arg_n)$

$\mathbf{A}($(proc $\varepsilon$ ($var_1 \ldots var_n$) $expr_1 \ldots expr_m$)$, arg_1 \ldots arg_n)$
     let  $\varepsilon^* = \varepsilon \cup \{var_1 \mapsto arg_1 \ldots var_n \mapsto arg_n\}$
    $\mathbf{E}(expr_1, \varepsilon^*)$ ;
      ...     ;
    return $\mathbf{E}(expr_m, \varepsilon^*)$
```

Function **A**, called `apply` in Appendix A, performs the call to procedure or primitive fn with arguments *exprs*. If the expression fn evaluates to a primitive procedure *prim*, **A** calls the primitive with the argument values *args*, returning the primitive's return value. Section 6.6 further describes primitive procedures.

When fn evaluates to a procedure value function **A** binds the variables from the formal parameter list $var_1 \ldots var_n$ with arguments from $arg_1 \ldots arg_n$ and produces an extended environment ε^* by extending environment ε captured in the procedure value with these bindings. **A** then evaluates the body expressions *body*, using the extended environment ε^* as the local environment. The extended environment ε^* is in effect only for the duration of the evaluation of the body expressions.

The procedure body may contain multiple expressions $expr_1$ ... $expr_m$, which all evaluated in sequence. Body expressions $expr_1$ ... $expr_{m-1}$ are evaluated only for the side-effects that may occur during their evaluation, their return values are discarded. The procedure call returns the value produced by evaluating the last body expression $expr_m$.

[3]In fact, this strict prescription of order of evaluation is a deviation from the Scheme standard, where arguments are evaluated in unspecified order.

6.5.2 Variables and environments

```
var G = '([cons . <primitive cons>]
          [car . <primitive car>]
          [cdr . <primitive cdr>]
          ...)
```

The SensorScheme interpreter accesses variables bound in environments. SensorScheme uses two distinct environments, a local and a global environment. While their role is somewhat different, their structure is identical.

Just like expressions, environments are constructed of collections of SensorScheme values. Environments have the structure of an *association list* – a list containing pairs that each hold a key and a value.

As shown above, **G** is initialized at startup with bindings to the *primitive procedures* available to the interpreter, stored as an association list.

The definitions of **E** and **A** used the following operations on environments:

$\varepsilon[var]$

Variable reference denoted as $\varepsilon[var]$ looks for the value bound to a variable var in environment ε. This is achieved by traversing the environment from first to last until a binding containing variable var is found, and the bound value is returned. Function `lookup` in Appendix A implements this behavior.

$\varepsilon[var] \leftarrow value$

The **`set!`** primitive expression replaces the bound $value$ to a variable var present in an environment ε using notation $\varepsilon[var] \leftarrow value$. Again the variable binding is looked up by traversing the environment. The found pair holding the binding is then altered, replacing the value in its `cdr` with the new value. Function `update!` in Appendix A implements this behavior.

$\varepsilon \cup \{var \mapsto value\}$

Extending an environment, denoted by $\varepsilon \leftarrow \varepsilon \cup \{var \mapsto value\}$ places a new binding at the front of the environment. Definitions extend the global environment and procedure applications extend the local environment. The primitive `cons` is used in Appendix A to add a binding to the front of the environment.

6.5.3 Program loading

As described in Section 6.2, a SensorScheme program consists of a sequence of definitions and expressions contained within a module and the library modules it refers to. When wirelessly transferring a program these definitions and expressions are analyzed and collected in an *injection* message (with message type symbol `inject-handler`) and transported to a sensor node, where the interpreter executes the expressions in the message. During the analyzation process, macros used in the external representation are expanded into more basic expressions.

Loading and execution of the program expressions contained within an injection message consists of subsequent evaluation of the expressions in the `inject-handler` message according to the following definition of the `inject-handler` message handler:

```
(define-handler (inject-handler expr-ls)
  (for-each (lambda (e) (eval e empty-env)) expr-ls))
```

All expressions in the message are evaluated first to last using `eval` with an empty local environment. The global environment contains the set of primitive procedures configured in the interpreter, and possibly variables **`define`**d earlier. Those expressions that are definitions – `(`**`define`** *var*`,` *value*`)` – load the application by creating the global variable or procedure *var*. Other, non-definition expressions initialize and start the application, for example by starting a timed infinite loop, as shown in Chapter 5.

6.6 Primitives

Primitive procedures or *primitives* are procedures accessible to SensorScheme programs that are implemented with native processor instructions. Primitives implement operations on basic types and provide access to the operating system and hardware resources. Section 6.4 already described the operations on basic types and Chapter 5 introduced some of the primitives to access WSN-specific functionality.

6.6.1 Basic operations

Operations on basic types include type predicates `number?`, `symbol?`, `pair?`, `boolean?` and `null?`, the equality test `eq?` and arithmetic and comparison operators `+`, `-`, `*`, `/`, `modulo` and `<`, `<=`, `=`, `>`, `>=`. Additionally, there are the operations on *pairs* `cons`, `car`, `cdr`, `set-car!` and `set-cdr!`. All of these operations are implemented as primitives, with semantics identical to standard Scheme.

Additionally, continuations, discussed in Section 5.9, are made accessible to SenorScheme programs through the `call-cc` primitive.

6.6.2 Eval

SensorScheme implements `eval` as a primitive.

It uses the SesnorScheme interpreter to evaluate an expression according to the description in Section 6.5, with an empty local environment, and returns the value calculated by the interpreter.

The operation of SensorScheme's `eval` primitive differs somewhat from its counterpart in standard Scheme, which accepts an expression in external representation, where expressions include macro-invocations, to be expanded before evaluation begins. SensorScheme `eval` does not macro-expand its argument expression; macro-expansion occurs only at the gateway.

SensorScheme uses `eval` as an essential part of its operation, as the means of loading and executing programs on sensor nodes. Besides `eval` SensorScheme includes an `inject-handler` *message handler* primitive (discussed further in Chapter 7) which invokes `eval` for each of the expressions in the message, as described already in Section 6.5.

6.6.3 WSN primitives

Besides the general purpose primitives described above, that are part of the Scheme standard, SensorScheme contains a set of primitives unique to SensorScheme to access and control WSN operating system and hardware.

`now`

The first one, `now`, returns the current time as 1/16 seconds since the device started. Its primary use is in conjunction with the primitive `call-at-time`.

`call-at-time`

Typical operating system functionalities such as starting a new process or thread of computation, as well as signaling some amount of time has passed (usually referred to as a *timer*) are all implemented with the primitive `call-at-time`. It accepts two arguments, the first one the time t at which to schedule a computation, the second a function f of one parameter that is called when the specified time has arrived. `call-at-time` maintains a queue (built as an association list made up of *pairs*) where the scheduled functions are added and removed when their scheduled time has arrived. At the scheduled time t function f is called with t as its single parameter.

The `call-at-time` primitive is usable for different scheduling tasks. First, it can be used to implement a timed delay loop as the applications in Chapter 5 show. This use is similar to repetitive timers as provided by WSN operating systems such as TinyOS.

Second, `call-at-time` can be used in the place of the one-shot timers that TinyOS and other operating systems provide. Section 5.9 shows the use of `call-at-time` to wait for timeouts of the reliable communication protocol.

Third, by scheduling a computation at the current time, `call-at-time` essentially spawns a new thread of computation. Applications can use multiple threads, each of which may be blocked waiting for I/O, at which time the other threads can resume their operation.

SensorScheme uses coarse-grained time measurements of 1/16 seconds for two reasons. First, with a 31-bit integer as data type, time values overflow after 4.25 years. While this might not cover the lifetime of a deployed sensor network, it is a long enough duration for many sensor practical situations. Second, not allowing more fine-grained timing prevents excessively high computational loads resulting from high frequency delay loops. Typical WSN applications use low frequency sampling and do not have any need for sub-second scheduling. From our experience, a time resolution of 1/16 seconds is sufficient for WSN applications.

Sensing

Access to sensors is crucial for wireless sensor networks. SensorScheme provides primitives to read sensors, of which we have seen examples like `sense-temp`. The sensors available is hardware platform-specific, but many senors have a similar interface. Primitive `sense-temp` and similar primitives accept no parameters and return a sensor measurement as a *number*.

Some sensors do not return a value immediately, but take some time, for example when they are connected through an ADC (analog to digital converter). TinyOS uses a split-phase approach to read these sensors, with a `read` command to initiate the sensing, and a subsequent `readDone` event that returns the result when the sensor has produced a value. In SensorScheme these sensors still have just a single `sense-*` primitive that returns the sensed value. These primitives store a continuation of the current computation, then call the TinyOS `read` command and exit the SensorScheme interpreter. When the `readDone` event is signaled it calls the stored continuation to continue the SensorScheme evaluation. According to their split-phase nature, between `read` and `readDone` other computations can take place, including SensorScheme evaluation such as handling received messages and scheduled timers.

Other 'sensors' only signal the occurrence of events, such as the push of a button. These sensors are implemented in SensorScheme in a similar manner to split-phase sensors: applications call the sensor primitive (such as `button-pushed`) to express interest in the event of a pushed button. The primitive then blocks the computation while storing the current continuation. When the button is pressed, the computation continues with a call to the stored continuation.

Output

WSN hardware platforms have only little support for user output. Commonly, the platforms have some LEDs to show some program state, and a serial port that can be used to output text for debugging purposes. SensorScheme provides the primitives `blink` and `print` for these purposes. Primitive `blink` accepts a single number and uses the lowest significant bits to set the state of the array of LEDs on a sensor node. Primitive `print` accepts any number of arguments and writes these in external representation to the node's serial port.

Communication primitives

Communication in SensorScheme takes place through *communication primitives*. Chapter 7 describes SensorScheme's communication method and primitives.

6.6.4 Implementation

Each primitive is implemented as a module for nesC, the implementation language of TinyOS, and are accessible by the interpreter through a NesC interface. Different kinds of primitives exist. Most primitives, performing simple arithmetic operations, are accessed through the `SSPrimitive` interface. Listing 6.1 shows the NesC interface definition, as well as the implementation of one of the primitives, `AddPrim`. The interpreter calls a primitive using the `SSPrimitive.eval` command. The `AddPrim` primitive in Listing 6.1 access its arguments using the `arg1` and `arg2` C macros, performs a type conversion and type check `C_numVal`),

```
interface SSPrimitive {
  command ss_val_t eval();
}

configuration AddPrim {
  provides interface SSPrimitive;
}
implementation {
  components AddPrimM;
  components SensorSchemeC;

  SSPrimitive = AddPrimM;
  AddPrimM.SSRuntime -> SensorSchemeC;
}

module AddPrimM {
  provides interface SSPrimitive;
  uses interface SSRuntime;
}
implementation {
  command ss_val_t SSPrimitive.eval() {
    int32_t x = C_numVal(arg_1);
    int32_t y = C_numVal(arg_2);
    return ss_makeNum(x + y);
  };
}
```

Listing 6.1: NesC interface and implementation of a simple primitive. `C_numVal` and `arg_1`, `arg_2` are C macros facilitating access to and type checking of arguments.

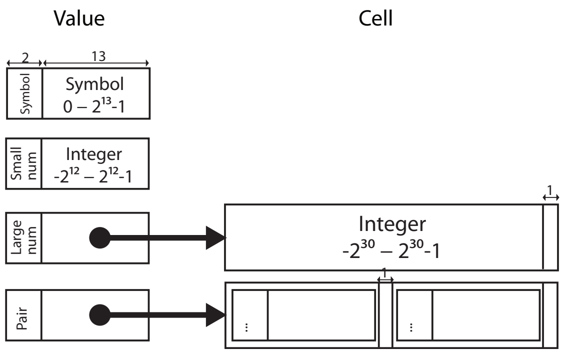


Figure 6.2: Memory layout of SensorScheme values.

computes the result, and returns the calculated result as a SensorScheme value (`ss_makeNum`).

6.7 Implementation

6.7.1 Data types and memory

We have argued in Chapter 4 to use a memory pool consisting of small, equally-sized blocks. Blocks may contain references to other blocks, but references must be distinguishable from other values to enforce memory protection. The SensorScheme language and interpreter has been designed entirely to support this model of memory allocation.

As described in Section 6.4, SensorScheme distinguishes the data types *number*, *symbol*, *pair*, *boolean* and *null*. Symbols have only a numeric representation in memory, and typically less than 50 symbols are used in any WSN application. The boolean and null types consist of only three distinct values. these are represented as special cases of symbol values. This leaves only three data types to be represented in memory.

Figure 6.2 shows the memory layout of the *number*, *symbol* and *pair* data types. The memory pool from which SensorScheme memory is allocated contains *cells* of 4 bytes in size. Cells are used to store pairs as well as large numbers.

SensorScheme uses *values* of 15 bits in size. A *value* contains both its type as a *type tag* and content in a *value/address* field. A value's *type tag* occupies the least significant two bits. There are four type tags, corresponding to the data types and their encoding in memory.

Symbol
: The value bits contain the symbol's numeric representation.

Small number
: For numbers n, $-2^{12} \leq n \leq 2^{12} - 1$, the value/address bits contain the numeric value.

Large number
: All numbers n outside the small number range and $-2^{30} \leq n \leq 2^{30} - 1$ are stored in a separate cell, the address of which is contained in the 13 value/address bits.

Pair
: The value/address bits of a pair contain the address of the cell where the pair is stored. The cell contains two 15 bit regions that contain the pair's *values.*

The number data type is represented by either a *small number*, for numeric values that can be contained in 13 bits, or a *large number* for values up to 31 bits in size. A *value* of type *large number* contains a reference to the cell containing the numeric value.

Addresses of cells in values and pairs are 13 bits in size, addressing at most $2^{13} = 8192$ cells, that occupy at most 32k bytes of RAM. None of the currently-available low power sensor node platforms provides this amount of memory.

6.7.2 Garbage collection

The SensorScheme language facilitates allocation of memory cells through the use of the `cons` primitive, or when creating a *large number.* To prevent running out of free cells quickly, when values stored in cells are no longer needed in future computations, they may be reclaimed for reuse.

To automatically reclaim unused cells we have implemented a simple two-stage reversed mark-and-clear garbage collection algorithm. First, bit *31* or the *free bit* is set on all cells, indicating a possible free cell. Then starting from the roots all live cells are traversed using the Deutsch-Schorr-Waite marking algorithm [SW67]. This needs only 1 additional bit per pair (bit 15) and no extra storage for long numbers. When a new cell is allocated, the cell pool is searched upwards using a free-pointer, until the first free cell is found as indicated by the free bit. The free pointer is assigned this cell's address and this cell is marked used (by clearing the *free bit*) and returned. After a number of subsequent allocations, when the free pointer points at the top of the store, a new garbage collection is started.

On MSP430-based platforms (such as the TMote Sky, our primary development platform) the implementation of this simple collection method has a running time of $12n + 52m$ CPU cycles, where n is the total number of cells and m the number of used cells. With a typical number of 2048 cells, a garbage collection takes about 10 ms to execute when memory is half used.

6.7.3 Tail calls

SensorScheme is properly tail-recursive, just like standard Scheme [ADH+98]. Tail calls are procedure calls occurring at the *tail* of an outer procedure, where the outer procedure immediately returns the return value of the inner, without performing any computations between the tail call and the procedure end. Properly tail-recursive implementations support an unbounded number of *tail calls*, requiring no memory (on the *call stack*) per tail call.

6.7.4 Call Stack

Calls that are not tail calls save the context of the call and and local environment at the time of the call on a call stack. Each call frame on the stack consists of a linked list containing values to be preserved, and subsequent call frames are chained by pairs. Consequently, stack size is not fixed but can grow as needed by the application. When a call returns the last frame inserted from the stack is released, and saved values are retrieved from the stack. The pairs forming the last frame will no longer be accessible through any reference, and will be recycled at the next garbage collection.

6.7.5 Continuations

Retrieving the current continuation captures the call stack and local environment and makes these available as a first-class value. The SensorScheme call stack is implemented as a chain of pairs allocated from the cell pool. SensorScheme allocates stack space only as much as needed, thereby removing the stack memory allocation issues of multithreading. Because stack frames are allocated from the cell pool, an unbounded number of coroutines can be used (as long as free memory is available), without wasting memory on unused stack space as is the case with the thread libraries referred to earlier.

6.7.6 Optimizations and extensions

The design and implementation of the SensorScheme compiler described in this chapter can be modified to improve its performance. We describe a number such optimizations here, some of which have been implemented and are used in the evaluation experiments described in Chapter 9.

De Bruijn index notation

The local environment as described in Section 6.5.2 contains bindings of variables to locations, and is implemented as an association list. Local variables in programs always refer to a binding that is in a constant depth in the association list. It is therefore equivalent to refer to the depth in the environment, rather than variable symbol to which the variable's value is bound, in a manner similar to De Bruijn index notation for the lambda calculus [DB72]. Using this, lambda expressions need to contain only the number of formal variables they use, rather than a list of $formals$ indicating their names.

To implement this optimization, all local variable names need to be replaced by a numeric value indicating the depth of the environment where the variable's value is stored. The SensorScheme analyzing process (see Section 6.2.1) replaces the symbols of local variables with one of the special symbols `%l0` – `%l15`. The benefit of this optimization is reduced memory use. Lambda expressions are smaller, occupying fewer pairs in memory and during injection (see Chapter 7), and environments use fewer pairs to contain local variables. Note that this optimization is restricted to the local environment and does not apply to the global environment.

Cell pool in Flash memory

The description of the SensorScheme interpreter in this chapter assumes the cell pool is located in RAM only. In current WSN hardware platforms RAM is smaller than the on-chip Flash memory that stores the binary code image. Locating a portion of the cell pool in Flash memory may conserve valuable RAM. Values in the cell pool that will never change, such as program expressions may be stored in the Flash cell pool.

This optimization is not equally suited for all current low power sensor nodes. Devices using a Harvard architecture CPU, such as the Atmel AVR in the various Mica* devices cannot implement this optimization without significant loss in speed, due to the separate address space and instructions for access to RAM and Flash. The MSP430 used in the TMote Sky – our development platform – does support this optimization without loss of computational efficiency.

Address space and type extensions

The memory layout described here is able to use at most 32 KB of memory. Most high speed sensor node platforms have more RAM available than this. It is possible to extend the SensorScheme address space without change in behavior of the interpreter by using a different memory layout. We have not implemented an alternative memory layout for such devices, but simply note the possibility here.

Larger memory sizes also reduce the risk of memory fragmentation, suggesting the possibility of alternative allocation strategies than the uniformly sized cell pool described here. Alternative representations also open up the possibility of supporting more data types than only integer numbers, symbols and pairs, to include vectors and strings and non-integer numeric types such as floating point numbers. These extensions and optimizations have not been implemented, we merely mention them here as possible future work.

6.8 Differences to standard Scheme

The execution semantics of SensorScheme described in this section match the R^{5}RS Scheme standard wherever possible. SensorScheme has deviated from the standard to accommodate the nature of wireless sensor networks and the tight memory restrictions of WSN hardware platform. Wherever possible, we have sought to maintain conventions and requirements of standard Scheme. This section describes the main differences between SensorScheme and the Scheme standard.

6.8.1 Types

First and foremost, SensorScheme uses a reduced set of basic types. This set of types enable memory allocation of only fixed sized cells, to prevent memory fragmentation. These cells are small, only 4 bytes, to keep memory consumption within the strict limits of WSN hardware platforms.

6.8.2 Module system

The R^5RS Scheme standard does not include a definition of a module system, but most implementations of the standard do. In this spirit, SensorScheme implements its own module system. The module system is designed for use with sensor networks, by selecting only those procedures and primitives that are actually required to execute a program. This enables programs to execute on small memory footprint devices such as sensor nodes, making sure no memory is wasted on unneeded functionality.

6.8.3 Primitives

Related to the module system is SensorScheme's concept of *primitive procedures*, unique to SensorScheme. Primitives explicitly describe the functionality that must be implemented in the interpreters implementation language – nesC, and make it accessible to the module system. R^5RS describes only *built-in procedures* which include all of the languageÕs data manipulation and input/output primitives.

6.8.4 Description of semantics

Rather than describing the semantics of the SensorScheme language we have defined SensorScheme by the translation of source-files into expressions evaluated by the interpreter and the behavior of the interpreter.

Furthermore, the interpreter is defined more concretely. Section 6.5.1 defines the seven primitive expressions handled by the interpreter. The Scheme standard, by contrast, defines a larger set of syntactic forms that all must be supported by an implementation, leaving undefined which of these is primitive and which may be translated to primitive forms using macros.

Similarly, SensorScheme defines a concrete definition of the interpreter and environment data structures, instead of a more abstract description of the behavior of Scheme expressions.

6.9 Discussion

SensorScheme has a lot in common with virtual machine architectures, of which a number have been developed for wireless sensor networks. Maté [LC02] is an early virtual machine for wireless sensor networks, developed specifically for early classes of low power sensor nodes. Perk [Cor08] and Darjeeling [BLC09], as well as NanoVM [nan] are Java VM implementations for low power sensor nodes. Several Scheme and LISP implementations for embedded systems also use a virtual machine to execute programs [DF05, FD03, WO05, Bur02].

All share the goals of efficient reprogramming, platform independence and protection at the expense of slower execution speeds. The design difference between SensorScheme's interpreter and virtual machines do, however, have impact on various aspects of their behavior.

6.9.1 Program representation

A major difference between virtual machines and SensorScheme is the way in which program code is represented. Virtual machines represent programs as arrays of instructions, whereas SensorScheme uses chains of its native data types of pairs, symbols and numbers to represent programs.

Stack-based virtual machines such as the Java VM use a number of different kinds of instructions: operations on primitive types, load and store operations, jump and call operations. SensorScheme includes the functionality of each of these classes of instructions, represented in a different way, which we discuss here.

Primitive operations perform operations on primitive types such as arithmetic and logic operations and may include other operations (such as I/O or operating system functions) that are available as VM instructions. They are the virtual machine equivalent of SensorScheme's primitive procedures.

Load and store operations exist in varieties, depending on the location of the loaded or stored value. For example, the Java virtual machine distinguishes between values stored on the stack, in local variables, as fields of objects, in the constant pool and immediate arguments, and different instructions exist to load or store from these locations. SensorScheme loads and stores values only as variables in the local or global environment, or as constants or quoted values embedded within the program. SensorScheme implements each of these with a single primitive expression. Other operations, such as passing function arguments and return values are implicit in the structure of SensorScheme expressions, rather than implemented as load and store operations.

Jump and call instructions have equivalents in SensorScheme's `if` and procedure call primitive expressions. Jump and call instructions contain an address to the target location. The bit width of these addresses determine the number and size of functions. For example, Maté's instruction format supports a maximum instruction array length of only 256 instructions per function, and only a very limited number of functions. Similarly, the Java VM limits methods to a length of 2^{16}. SensorScheme's procedures and expressions are values stored in memory cells and have no size limitation other than the interpreter's address space.

Conclusion We reviewed the difference between SensorScheme's program representation and the more common alternative – stack based virtual machines. For the different kinds of functions performed, as instructions or otherwise, SensorScheme's program representation is both simpler and more versatile than the use of virtual machine instructions. The multitude of instructions on VM platforms, usually 256 or less, is reduced to just seven primitive expressions and a variable number of primitive procedures. Additionally, the structure of SensorScheme programs – passing parameters to functions, the order of computations and the end of procedures – all express the operation of a program, without being explicitly encoded in instructions, as is the case for virtual machines. Subsequently, SensorScheme uses just a single type of variable store – the environment, compared with a wider variety in object-oriented and stack-based VMs.

The causes pointed out above contribute to SensorScheme's suitability for WSN platforms: its simplicity of design enables a compact implementation. Evaluations later in this chapter will further evaluate this hypothesis. The question remaining from these observations is whether the simplicity of SensorScheme's runtime environment will have advantageous or detrimental effects for the runtime speed and memory use, which we will also address in this chapter.

6.9.2 Configurability

SensorScheme interpreters can be configured by a user by choosing the set of primitive procedures included in the interpreter installed on sensor nodes. Similarly, when using a VM design, virtual machine instructions can be made configurable, as Maté shows. Maté supports the customization of its virtual machine – to make it 'application-specific' [LGC04] – through the use of a configuration file stating all the instructions supported. Different configurations are associated with different source languages, and may contain language-specific instructions, as well as hardware platform specific instructions. Maté uses a byte-code instruction format, allowing at most 256 different instructions. For optimal efficiency and code size, it is customary to specify exactly 256 instructions, for example by including multiple constant loading instructions. Selecting the set of instructions in a configuration therefore is a manual undertaking. Furthermore, the source language's compiler requires knowledge of the selected set of instructions to properly compile a program.

SensorScheme's primitive selection procedure is fully automated and implicit in the module file used to compile an interpreter. Primitives correspond directly to procedure calls in the program text, so no explicit support is needed from the compiler to generate calls to the primitive procedures.

6.9.3 Memory safety

Virtual machines and interpreters provide a virtualized execution environment, restricting memory access only to valid locations, and controlling I/O operations. Additionally, each value is annotated with a type tag, both in memory and during communication, and used by primitive procedures to check for validly typed operands. Together, this protects the underlying hardware from buggy or malicious programs. Virtual machines like the Java VM or Maté by themselves are not completely secure; only verification by static analysis of code can guarantee this level of safety.

The Java platform defines a byte-code verification process needed to ensure memory safety. Java's byte-code verification is a check for certain invariants of the program code that needs to take place before the program is run. Byte-code verification is a computation and memory intensive process beyond the capabilities of low power sensor nodes. Resource-lean Java platforms (such as those on mobile phones) use offline verification and cryptographic code signing. This in turn requires a cryptographic library in place on the Java runtime platform and a key-exchange and validation infrastructure.

Java's need for byte-code verification arises from the fact that instruction and method arguments on the stack need to be of the appropriate type and number to ensure safety. It is not trivial to infer the code locations where those arguments have been placed on the stack, hence the need for a separate process to verify this invariant.

SensorScheme's execution model obviates the need for verification or code signing. All operations in the interpreter check the data type of its arguments and do not depend on the type and structure of data that it does not check right before execution. For example, SensorScheme's call mechanism collects procedure arguments into a list that is checked for the proper length during procedure and primitive calls. Furthermore, SensorScheme's dynamic typing forces explicit checks on the proper argument types of procedure arguments.

6.9.4 Memory and data types

The design of SensorScheme is based on a simple memory model and type system, where all memory is allocated as equally-sized cells. Compared to a more extensive type system such as Java's this has certain advantages as well as drawbacks.

Strings and numbers

SensorScheme's type system does not include a character or string data type. While this may seem an undesirable omission, the application scenarios do not require the use of strings. In places where unique identifiers are needed, symbols suffice. The absence of a display on the devices removes much of the need for manipulation of characters or strings on sensor nodes. A lack of string manipulation functionality reduces the binary footprint of the interpreter.

SensorScheme uses 31-bit integers as the only numeric data type. Smaller numbers are stored as 13-bit *small* numbers for memory-efficiency. Still, only 31-bit arithmetic is used for both the large and small integers. SensorScheme does not make use of floating point arithmetic. Again, the presented application scenario's do not require the use of a more extensive set of numeric data types. The simplicity of the presented design makes it possible to keep the implementation within limits of the used hardware platforms, and enables the use of the simple and efficient memory allocation techniques used.

Structs or Objects

SensorScheme does not include any compound data types except the *pair*, used to construct lists. As mentioned earlier, it is possible to write an object system closures, and the SensorScheme distribution contains one. The presented example applications do not, however, have great need for this, so we have omitted examples of its use.

Arrays vs. lists

Probably the most important omissions from SensorScheme's type system are arrays. The behavior of these can be implemented using linked lists, but at the extent of slower access times of $O(n)$, with n the list index of the accessed value, instead of $O(1)$ for arrays. Program memory in sensor nodes is limited, so n will never be very large, thus limiting the maximum access time.

The use of linked lists may also consume more memory than when arrays are used, especially when the arrays in use have a fixed size, known at compile time. When the number of items to store in an array is variable, reallocation of a larger array size may be needed, which temporarily increases memory use to more or less double the array size, and adds computation time for reallocation and copying. Compared to algorithms that incrementally increase the size of an array or list, the use of linked lists is not necessarily slower and consuming more memory, but depends on its pattern of use: how many reallocations do occur, what is the size of the array.

We have shown in the example programs in Chapter 5 that algorithms using linked lists are a natural fit to WSN applications. The applications repetitively append a list with a single item, and traverse the entire list once before discarding it.

Naturally, linked lists may also be used in virtual machines, for example by creating a Java class `Pair`. These `Pair` objects will, however, use more memory than SensorScheme's pairs, as a result of Java's architectural requirements (such as a reference to an object's *class*).

Code size

The use of expressions as linked lists rather than instruction arrays will similarly have a negative effect on the size of program code in memory. Flash program memory is, however, not the most restricted memory in WSN platforms. (The programs in appendix B do occupy only a small portion of the 16 KB available.) Additionally, SensorScheme's lack of strings and user-defined types reduce code size by not including type definitions and string literals, which are present in Java runtime platforms.

Automatic memory management

Platforms that ensure memory safety will need to include an automatic memory management mechanism. Some form of garbage collection is used for the various Java platforms as well as SensorScheme and other embedded Scheme or Lisp implementations (as opposed to reference counting in use by the Python language). While a wide variety of garbage collection algorithms exist [WJNB95], all have in common that runtime performance is linked to the amount of memory available. In situations where a large fraction of total available memory is permanently in use, after a garbage collection, only little memory is free and garbage collection will occur again frequently. This may happen when a program stores large data structures or keeps cached information. SensorScheme dynamically allocates all internal interpreter data, including environments and stack frames, and is particularly reliant on free memory to achieve good runtime performance. Other systems with high allocation rates, such as the Darjeeling Java VM [BLC09] dynamically allocate stack frames, and execution speed will thus be similarly sensitive to the free space available.

Symbols

SensorScheme uses *symbols* as a primitive data type, inherited from the Scheme language. Symbols serve as unique, static identifiers shared by the entire network. This is an important asset in any distributed system, needed to access resources that are shared by devices or accessible by other devices. Symbols remain semantically meaningful across communication, and are used as identifiers of data or code on remote nodes: symbols are used as protocol tags, identifying program code to be executed on arrival of the message.

In the same way, symbols play a role in linking different parts of a program. Programs refer to procedures and data through symbolic names, rather than in-memory addresses. When dynamically loading programs, some form of symbolic linking of program parts (or modules) is required. The module-loading WSN operating systems described in Chapter 2 use some form of symbolic linking between modules.

While symbols are identifiers with a textual representation, SensorScheme's implementation of symbols have only a numeric representation on the nodes in the network. This way, symbols take up as little as 10 bits in network representation and only 16 bits in memory, which makes symbols a suitable data type for all of its roles.

6.9.5 Conclusion

SensorScheme's design is focused on simplicity, evident in the small type set of data types supported. The use of pairs and symbols to represent code, instead of arrays of instructions makes the interpreter smaller and more simple. Still, the design easily scales to larger platforms when required in the future. Memory safety can also be guaranteed better. SensorScheme's compact design does have potential drawbacks in terms of increased computation time and memory use, but these are not likely to cause significant effects in WSN applications.

Chapter 7

Communication

In the previous chapters we have discussed the design of SensorScheme with regard to the way it represents and executes programs, and shown how to use the language and execution environment to build sensor network applications. Until now, we have, however, left out a description of the communication mechanisms employed by SensorScheme.

This chapter describes ObjectStreams, the communication mechanism for the Sensorscheme platform. It has been designed to serve the communication needs of the platform to transport program code as well as application data.

ObjectStreams is itself not a communication protocol such as the routing and transport protocols described in Section 2.5.1. Instead it builds on communication protocols for wireless sensor networks such as broadcast, tree collection or dissemination protocols as we have reviewed in Section 2.5. It is a mechanism to transport SensorScheme data structures in serialized form in a stream of multiple packets using the transport mechanisms of these routing and transport protocols.

In line with SensorScheme's goals, ObjectStreams is a communication abstraction designed to simplify the task of writing WSN applications by minimizing the amount of code to write, and abstracting from protocol- and platform-specific details.

```
  (define-handler (demo-msg sensor-val lst)
2   (blink sensor-val))

4 (bcast (msg demo-msg (sense-mag) '(1 2 3 4 5 6 7 8))))
```

Listing 7.1: broadcast example program

7.1 Requirements and use

We start this chapter with a review of our communication method in the way we have used it in Chapter 5. Subsequently we present in this section a number of properties and requirements that enable implementation of the presented method on WSN platforms.

The example applications have shown already that sending a message is similar to a remote procedure call. Some minor adjustments to the syntax (by defining macro's) make the definition of a message handler akin to a regular procedure, and transmitting a message similar to calling that procedure. The contents of a message are the arguments to the call of the remote procedure.

The listings in Chapter 5 showed the use of primitive `bcast` to broadcast messages, also shown in Listing 7.1. (In fact, `bcast` is not a primitive, but a wrapper procedure around primitive `send-local`, but for simplicity we assume here that it is.)

Procedure `bcast` takes only one argument, which is the message to communicate. All examples use *macro* `msg` to construct a message.

`(`**`msg`** $msgsym\ value_1\ \ldots\ value_n$`)`

`==>`

`(list (`**`quote`** $msgsym$`)` $value_1\ \ldots\ value_n$`)`

Macro `msg` expands to create a message, which is a regular SensorScheme list headed by a symbol – the message type symbol *msgsym*, followed by content $value_1$ to $value_n$.

The `msg` invocation in line 4 above macro-expands to

```
(list (quote demo-msg) (sense-mag) '(1 2 3 4 5 6 7 8)).
```

Evaluation of this expression yields the following list:

```
(demo-msg 15 (1 2 3 4 5 6 7 8))
```

where sensing the magnetometer through `(sense-mag)` produces the value 15.

ObjectStreams broadcasts this list as a message to all neighbor nodes. Upon reception these nodes all produce an identical copy of the transmitted list in their own memory.

For each message that arrives the Sensorscheme interpreter *handles* the message as described in Section 5.3. The message's message type symbol `demo-msg` refers to a message handler on the local device.

The message handler definition, `define-handler`, is another macro, and expands as follows:

```
  (define-handler (demo-msg sensor-val lst)
     ...)

==>

  (define (demo-msg src sensor-val lst)
     ...)
```

As the expanded definition shows, a message handler is a regular procedure. The macro-expansion introduces the additional parameter `src`, that binds to the sending node's **`id`** when called. The handler's other formal parameters (`sensor-val` and `lst`) will bind to the message's contents (`15` and `(1 2 3 4 5 6 7 8)`).

In effect, handling a message is identical to a call to procedure `handle` in Listing C.1:

```
(define (handle src msg)
  (apply (eval (car msg)) (cons src (cdr msg))))
```

where `msg` is the received message list, and `src` the **`id`** of the node that broadcast the message.

Communication primitives (such as `send-local`) return a boolean indicating the error status of the communication. When the operating system reports that the transmission has not been successful **`#f`** is returned, and **`#t`** otherwise.

As Chapter 6 described, SensorScheme programs are represented as SensorScheme values: compound structures built from pairs containing other pairs or symbols and other primitive types. Such programs are themselves valid Sensorscheme values and may be transmitted in an Objectstreams message. The Injection phase described in Section 6.1 uses this method to inject a message into the network. It wirelessly transmits a program to a node as the contents of a message with message type symbol `inject-handler`. All nodes contain a primitive procedure called `inject-handler` that is the message handler for a message containing a program. It `eval`uates the program in the message content to start its execution, according to the procedure described in Section 6.5.3.

Requirements

To realize ObjectStreams' behavior described above we define the following requirements for its design and implementation:

Encode messages
: Messages take the form of values in memory with recursive references to other values. Packets transmit as their *payload* an array of bytes. Similar to the external representation (see Section 6.4) Objectstreams uses a *network representation* to encode the message into a sequence of bytes.

Multi-packet messages
: The content of an ObjectStreams message is not bound to a maximum size, and the length of a message, as it is encoded in network representation may exceed the size of a single packet transmitted by the radio hardware. In particular, SensorScheme programs are large enough to require transmission in multiple packets. WSN operating systems provide a communication interface that only deals with single packets, the size of which is determined by the radio hardware. ObjectStreams needs to make sure that a message transmitted in multiple packets arrives correctly on the receiving side: it must preserve the order in which the packets were sent, and detect if any of the packets were lost and optionally request retransmission.

Blocking operations
: The description of Sensorscheme communication above described the operation of communication primitives as a *blocking operation*: When a program calls `send-local` the primitive returns only after the transmission has been completed, either successfully or with failure. This is in contrast to the event-based interface of WSN operating systems (at least those without native support for threads). We have argued in Section 4.4 that blocking communication is preferable to event-based interfaces for the conciseness and readability of programs, and therefore ObjectStreams uses blocking communication. When messages are transmitted in multiple packets, the communication primitive must return only after all packets have been sent. In-between those transmissions, other operating system as well as SensorScheme tasks may run.

Multiple communication protocols
: Wireless Sensor Networks require the use of more than a single communication protocol, for different modes of communication. Simple broadcast, collection along a routing tree, dissemination from the gateway into the entire network are examples of these, and applications may need one or more of these simultaneously. ObjectStreams is not itself a transport or routing protocol, it is an application level mechanism to facilitate communication for SensorScheme programs using any of these protocols.

The following sections describe the techniques and mechanisms used to put these requirements into effect.

7.2 Design

The ObjectStreams communication mechanism consists of three parts. Each will be described in its own section below.

Serialization into network representation
: ObjectStreams transports SensorScheme values across the wireless network. Values are encoded into *network representation* – a serialized, flat encoding to be carried in one or multiple network packets discussed in Section 7.3.

Multi-packet sequences
: Messages may be carried in a sequence of packets. Correctly decoding the entire message requires all transmitted packets in the original order. Section 7.4 describes the packet header information and processing required to ensure this.

Communication primitives and protocols
: ObjectStreams is a general communication abstraction used to make lower-level communication protocols available to SensorScheme programs. Section 7.5 discusses how SensorScheme's communication primitives provide that crucial connection.

7.3 Serialization

ObjectStreams transforms the list that represents a message into a *network representation* by serializing the values in the list as a sequence of bytes. The

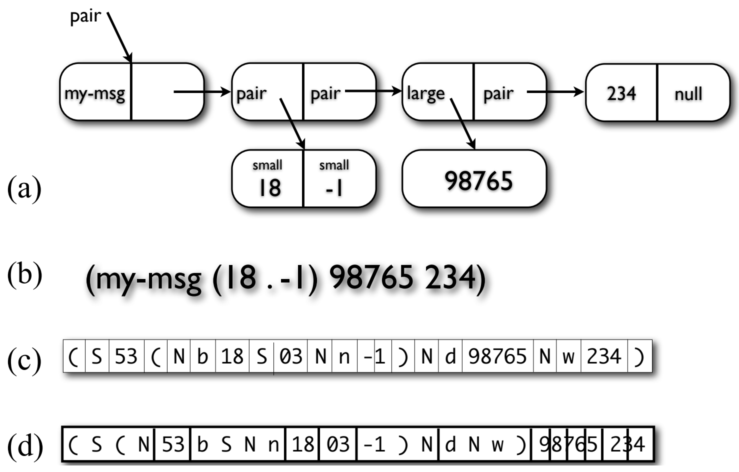

pair
my-msg
pair
pair
large
pair
234
null
small 18
small -1
98765
(a)
(b)
(my-msg (18 . -1) 98765 234)
(c)
(S 53 (N b 18 S 03 N n -1) N d 98765 N w 234)
(d)
(S (N 53 b S N n 18 03 -1) N d N w) 98765 234

Figure 7.1: Encoding of values into network representation

byte sequence is put into one or more packets, each filled to their maximum payload size, or with as many bytes as are left at the end of the sequence. When the packets arrive at a receiver, it reconstructs the message as a list of values in memory.

Figure 7.1 shows an example of the method of encoding values. We will describe the encoding process in reference to this example Figure 7.1 (a) shows the internal representation of a message in memory, and figure 7.1 (b) its external representation. The message is a SensorScheme list, with header symbol `my-msg` and three values. The first value is a pair containing two small numbers, denoted as an *improper list* (using the dot notation described in Section 6.4).

Figure 7.1 (c) and (d) show ObjectStreams network encoding, where instead of characters separated by whitespace, it uses a compressing encoding, emitting tokens and single bytes according to the algorithm in figure 7.2. In this notation symbol `my-msg` is replaced with its numeric representation – 53. (Only the numeric representation is present inside the interpreter, not the symbol's string representation.)

Figure 7.2 shows the encoding algorithm in pseudo-code that uses matching on the type of the value to define the operations performed in the main encoding routine – `encode` – and the auxiliary procedures `encode-num` and `r-encode`.

```
data Value    = Pair l r | Symbol s | Num n | Null
data ValToken = '(' | ')' | 'S' | 'N'
data NumToken = 'n' | 'b' | 'w' | 'l'

encode (Null)       = emit ('('), emit (')')
encode (Pair l r)   = emit ('('), encode (l), r-encode (r)
encode (Symbol s)   = emit ('S'), emit-bytes (s, 1)
encode (Num n)      = emit ('N'), encode-num (n)

r-encode (Null)     = emit (')')
r-encode (Pair l r) = encode (l), r-encode (r)
r-encode (s)        = emit (Int), encode (Symbol dot), encode (s)

encode-num ($-2^3 \le n < 2^3$)       = emit ('n'), emit ($n$ mod 4), emit ($n$ / 4)
encode-num ($-2^7 \le n < 2^7$)       = emit ('b'), emit-bytes ($n$, 1)
encode-num ($-2^{15} \le n < 2^{15}$) = emit ('w'), emit-bytes ($n$, 2)
encode-num ($-2^{31} \le n < 2^{31}$) = emit ('l'), emit-bytes ($n$, 4)
```

Figure 7.2: Algorithm for encoding of values into a token sequence.

Tokens encode which one of four possible types of data is encoded next, after which the actual data follows.

The message in Figure 7.1 is a list, and `encode` matches on the *Pair* clause. It emits an opening bracket token:`(`, the first token in Figure 7.1 (c). It then proceeds encoding the symbol `my-msg`, emitting subsequently an 'S' token (`S` in Figure 7.1 (c)), and a single byte containing the numeric value of the symbol – 53. Encoding continues to the right-hand side of the first pair using `r-encode`

Figure 7.1 (d) shows the result of serializing the example message into a sequence of 13 bytes. The individual bytes are surrounded by thick black lines. Encoding of both tokens and single- or multi-byte numbers proceeds in a manner similar to a technique in compression algorithms such as LZ77, part of the Zip compression program [ZL77]. Tokens are encoded using 2 bits, and four subsequent tokens are put together in a single byte. Bytes containing tokens alternate numbers in a single byte or more.

Encoding of values and filling packets proceeds in a streaming fashion: the value is traversed and tokens emitted until the packet's payload section is filled, or the entire structure is encoded. When a packet is filled before the entire structure is traversed, the current state of encoding is stored as a *continuation* (see Section 5.9) of the encoding process, to be used at a later moment when encoding can resume. When the next packet is available (this may be the same packet buffer after transmission has finished) the encoding continuation is resumed, and the encoding process continues, storing emitted tokens in the new packet.

Similarly, receiving nodes decode the contents of a packet sequence on a per-packet basis. During decoding, the encoded tokens are read from the packet payload, and the encoded message list is created by automatically allocating values as needed. When a message encoding uses multiple packets, at the end of each packet (except for the last packet of the sequence), the current decoding state is stored in a *continuation*, to be resumed when the next packet in the sequence arrives. When the last packet of the sequence is received, ObjectStreams returns to the interpreter the message contained in the packets as a SensorScheme list. When not all packets in the sequence are received after a certain time-out, the continuation of the packet last decoded is removed. SensorScheme's garbage collection process will reclaim the cells in the partially received message.

The network representation is able to encode tree-structured data only. When cycles exist in the message content, or multiple references to the same pair, this structure cannot be encoded. In case of cyclical data structures, the encoder will produce an infinitely long message as it continues to traverse the circular structure. Applications using ObjectStreams should avoid sending cyclical data structures.

Note that the encoding and decoding operations described above are lightweight both in memory use and computational requirements: During encoding, each value is only inspected once: the total number of values, and hence message size is not known prior to encoding; furthermore, the encoding and decoding operations ensure that only a single packet buffer is needed during both transmission and reception, to store or retrieve the values communicated; and finally, the maximum number of cells allocated during reception never exceeds the number of cells in the value structure sent.

7.4 Packet Sequencing

As the previous section described, ObjectStreams carries its payload in a sequence of packets. Receivers of this packet sequence must be able to recognize all packets belonging to the same sequence, and process them one by one, in order, without missing packets or duplicates. Additionally, nodes may receive multiple streams simultaneously, possibly even from the same sender. The ObjectStreams protocol defines a packet header format designed to take care of all of this, while being compact enough to introduce only marginal overhead. The encoded value structure can have arbitrary complexity and size, and its size can therefore not be found except by traversing the entire structure. The stream of packets containing the value structure encoding therefore does not contain information regarding the total number of packets it consists of; it is a stream of packets, marking only its start and end.

ObjectStreams payload is carried in a stream of packets. Objectstreams is implemented with TinyOS 2 and uses the TinyOS 2 Packets as the underlying protocol layer. TinyOS 2 *Packets* make use of a packet buffer that contains a header and payload, and is transmitted as a single unit by the radio hardware.

Besides TinyOS's header fields ⟨ *source address, destination address, packet length* ⟩ ObjectStreams defines three header fields: a start flag a, a stop flag z and a packet number m – implemented as a fixed-bit-width number of b bits (current implementations use either a 6 or 14-bit packet number). The start flag marks the start of a stream, and is set only on the first packet of the stream. Similarly, the end flag is set on the last packet of a stream. The packet number is used to track the order of packets in a stream. Every subsequent packet in the stream contains a packet number $m_{i+1} = (m_i + 1) \bmod 2^b$, until the last packet, which is marked with the stop flag.

The tuple ⟨ *packet number, source address* ⟩, called a *packet ID*, uniquely defines every individual packet. Only retransmitted packets use the same packet ID; otherwise no two packets with the same packet ID should be alive in the network at any moment. When a packet is received and processed at its destination it is no longer active and a packet with the same packet ID is permitted again. Packets can remain active for a longer duration because of buffering or caching at either the source or destination node, or on any intermediate node in the case of multi-hop link models.

Multiple streams between the same sender and receiver can be active at any one time, alternating packet transmissions. Streams are numbered S_0, S_1 ... S_n, each of which consists of packets p_0, p_1 ... p_m. To minimize the probability of duplicate packet IDs active in the network, the packet number of a stream's first packet is initialized to the packet number of the most recently sent packet from the stream last created at the same node minus a number $n (\bmod 2^b)$, where n is greater than the number of packets the network can buffer.

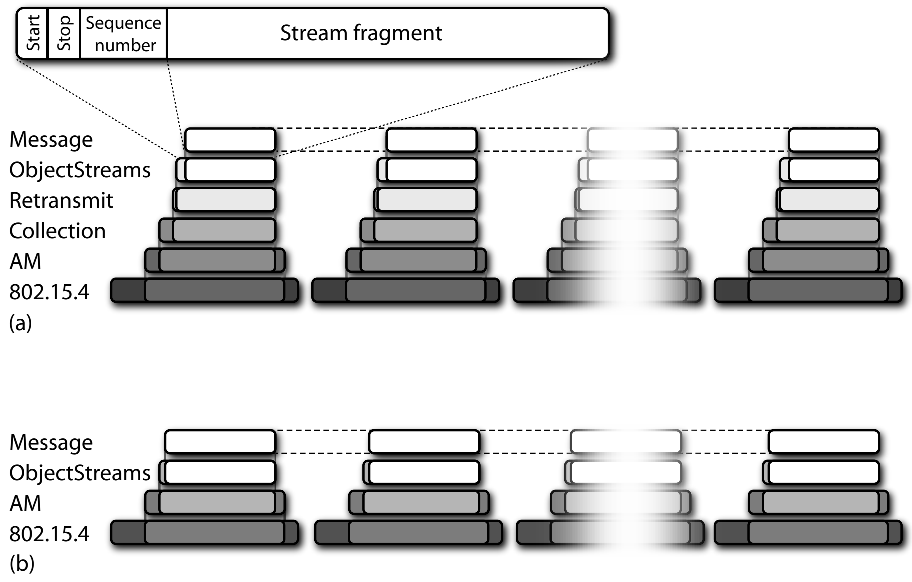


Figure 7.3: Position of ObjectStreams in the protocol stack

```
interface SSSender {
  ...
  command ss_val_t eval(am_addr_t *addr);
  command error_t send(am_addr_t addr, message_t* pkt, uint8_t *dataEnd);
  event void sendDone(message_t *msg, error_t error);
}

interface SSReceiver {
  ...
  event message_t* receive(message_t* msg, am_addr_t addr,
      uint8_t *data, uint8_t *end);
}
```

Listing 7.2: NesC interfaces for SensorScheme communication primitives

7.5 Communication primitives

The SensorScheme interpreter takes care of the serialization and sequencing described above, given a message as a SensorScheme list on the one hand, and packets on the other hand. Communication primitives connect the interpreter to an underlying communication protocol.

As we have shown already, SensorScheme supports simultaneous use of multiple communication protocols, each implemented using its own set of primitives. It is possible to support any of the wide variety of protocols developed for wireless sensor networks reviewed in Section 2.5 by configuring the interpreter to include the primitives that implement access to these protocols.

SensorScheme distinguishes different kinds of primitives. Besides the primitives described in Section 6.6 SensorScheme uses *sender* and *receiver* primitives for ObjectStreams communication. The SensorScheme interpreter accesses these primitives using the interfaces `SSSender` and `SSReceiver`, shown partially in Listing 7.2. Communication protocols may consist of both a sender and a receiver primitive, or only one of these, depending on the mode of communication. When nodes use a protocol only to send data outside the network, or to receive from outside the network, only either a sender or receiver primitive is needed. For example, the protocols used to inject programs into the network support only a receiver primitive, since nodes will only receive data from these protocols. A single communication protocol implementation may also provide multiple sender or receiver primitives. The SensorScheme implementation of the TinyOS 2 Collection protocol provides both a `send-parent` and a `send-gateway` primitive, each for a different mode of communication using the protocol's routing tree.

In TinyOS 2, protocols typically expose the `Send`, `Receive` and `Packet` inter-

```
interface Send {
  command error_t send(message_t* msg, uint8_t len);
  event void sendDone(message_t* msg, error_t error);
  ...
}

interface Receive {
  event message_t* receive(message_t* msg, void* payload, uint8_t len);
}

interface Packet {
  command void clear(message_t* msg);
  command uint8_t payloadLength(message_t* msg);
  command void setPayloadLength(message_t* msg, uint8_t len);
  command uint8_t maxPayloadLength();
  command void* getPayload(message_t* msg, uint8_t len);
}
```

Listing 7.3: NesC interfaces for low-level communication

faces (see Listing 7.3) to send and receive using the particular protocol and access the protocol's payload inside packets. See the TinyOS packet protocols documentation for a more detailed description of the communication process [Neta].

When a SensorScheme application calls a sender primitive it invokes the primitive's `SSSender.eval` function, similar to a call to a `SSPrimitive.eval`. The function returns a message to be sent to the interpreter. The SensorScheme interpreter captures a *continuation* at this point in the application's execution, where it is resumed after the message transmission has finished.

ObjectStreams now allocates a new packet from a memory *Pool* and encodes this message into the protocol's payload area provided by the `Packet` interface. When the packet is filled, ObjectStreams calls `SSSender.send`, which instructs the primitive to call `Send.send` to transmit the packet, and stores the application's computational state and the packet encoding state as a *continuation*.

Sending packets is a split-phase process, where `Send.send` only schedules a packet transmission and returns immediately, and the `Send.sendDone` event is signaled after the transmission has either succeeded or failed. The sender primitive in turn calls `SSSender.sendDone` which resumes the encoding process, and sends subsequent packets.

When the message has been fully encoded and all packets are sent, the the interpreter resumes the SensorScheme application by invoking the saved continuation, supplying it either a success or failure return value.

When a lower-level communication protocol receives a packet, the protocol's receiver primitive signals the `SSReceiver.receive` event to notify ObjectStreams of its arrival. ObjectStreams gathers all incoming packets and decodes them as they are received. As long as a message has only been received partially Objectstreams saves the decoding state in a continuation and keeps it until the next packet in the sequence arrives and continues decoding it. When the entire message has arrived, the interpreter finds the proper message handler procedure to further process it.

As an illustration of the use of ObjectStreams, we finish this section with descriptions of two example ObjectStreams transmission protocols, that we use in chapters 5 and 9 to implement and evaluate the example applications of Chapter 3.

7.5.1 Broadcast

The first ObjectStreams protocol implementation is positioned directly on top of the TinyOS 2 Active Message protocol to broadcast streams to other nodes within reach of the wireless connection (see figure 7.3 (b)). It uses unidirectional communication, to preserve the low cost of broadcast interactions between nodes. The sending node just broadcasts the stream's packets in sequence, without retransmissions.

When a broadcasted message is comprised of just a single packet, neighbor nodes just receive the packet, decode its contents – which produces a message in memory, and process the message with the appropriate message handler.

In case a message consists of multiple packets, a receiver must detect when a message has been lost, and decode each message directly as it arrives. The alternative strategy of buffering the packets until the last one has arrived will occupy multiple packet buffers in memory and is therefore not used.

Receivers can recognize the first packet in a sequence by a set *start flag* and subsequent packets by their subsequent *packet IDs.* Receivers detect missing packets by the arrival of another packet in the sequence or the absence of any more packets in the sequence after a certain time: After three packets have been received, if packet five arrives it means packet four has not been received. Similarly, if after packet three no more packets arrive within a set time limit (one second in the current implementation) the next packet in the sequence has been lost. The packet sequence ends when a packet with the *end flag* arrives. When a packet in the sequence is lost, the message cannot be properly decoded, so the entire message is discarded, and not delivered to the SensorScheme application.

Each packet is decoded immediately as it arrives, and its packet buffer made available for the arrival of other packets. Decoding the packet proceeds in reverse order of the encoding process, traversing the payload to find the tokens and numbers each packet consists of. As the tokens and numbers are decoded, the message structure is built in memory. When decoding reaches the last byte in the packet, and more packets follow in the current stream, the decoding state is recorded in cell memory along with the packet ID. Upon receiving the next packet in the sequence, decoding the message continues using the recorded decoding state, and so on, until the last packet has beed received.

Note that it is possible for a node to receive multiple messages simultaneously. For each of these messages the node allocates a sequence record, sets a timer, and stores the partially received message and decoder state. All of these data structures are allocated from the value cell pool, putting no restrictions on the number of simultaneously received messages, or the size of the message contents, except for the total amount of memory available in the cell pool.

The broadcast ObjectStreams protocol is the most straightforward ObjectStreams protocol. It does not retransmit lost packets, or buffer packets in the network, which may cause out-of order delivery. Nodes can send only one message at a time, starting a new sequence only after the previous has been completely transmitted. The bit-width b of the *packet number* header field can be small. This implementation uses a 6 bit packet number, making the total Objectstreams header just a single byte in size.

While straightforward, this implementation's lack of packet retransmission and buffering makes the protocol suitable primarily for broadcasting small, usually single-packet messages. Section 9.3 evaluates the performance of the intruder detection scenario using the *bcast* communication primitive, comparing it to other published implementations.

7.5.2 Reliable tree collection

The second implementation uses the Collection protocol in the TinyOS 2 distribution [Netb] to forward stream packets across multiple hops to a collection root (see figure 7.3 (a)). This implementation makes use of intermediate buffering and retransmissions to guarantee delivery of the entire stream.

The Collection protocol establishes a routing tree and assigns a parent to every node, which is a nearby node within communication reach. For the purpose of the collection protocol, nodes only communicate with their parent node. Encoding and decoding of messages and packets proceeds similar to the broadcast protocol described above. The ObjectStreams tree collection protocol uses a reliable transmission protocol to ensure all data will eventually arrive at the tree root. When a packet is transmitted, the receiving parent node immediately sends an acknowledgement message back to the sender. Our current implementation does this using the hardware acknowledgements of the IEEE 802.15.4 standard [IEE06]. When the sender does not immediately receive an acknowledgement, it assumes unsuccessful reception, and reschedules the same packet for transmission. When after three retransmissions a packet still has not been received, the node reselects a parent and attempts to send the entire message again.

Using this protocol, a message can be lost only in the case of a failing parent node, or lack of buffer memory to receive a message. When the transmission rate stays below the network's bandwidth capacity, reception failure due to insufficient buffer memory is very unlikely to happen.

Unlike the broadcast protocol, sending a message can take a considerable time, during which new messages may be available for transmission. Multiple messages may be sent concurrently, and the packet numbers of these packets must be disjoint. The bit width of the packet number header field b of this implementation is therefore chosen to be 14 bits, which makes the total header size 2 bytes.

The evaluation of communication in Section 9.3 tests the performance of reliable tree collection protocol in the environmental monitoring application scenario.

7.6 Discussion

This chapter described ObjectStreams, a communication abstraction for wireless sensor networks for communicating messages of larger size than a single packet payload. As the application scenario implementations in Chapter 5 suggest, the need exists to communicate more data at once than fits in a single packet. In current WSN platforms, it is common practice to construct communication protocols communicate using individual messages as provided by the underlying radio hardware, without the aid of an abstraction layer especially developed for communicating large payloads in multiple packets.

ObjectStreams facilitates communication of arbitrarily complex and large data. ObjectStreams is not a communication protocol, but an application-layer transport mechanism that can be used in combination with a multitude of communication protocols, which we have reviewed in Section 2.5. It serves to quickly and easily include communication in programs.

As a communication mechanism designed to communicate language-level values, ObjectStreams bears similarity to neighborhood abstractions such as *Hood*

[WSBC04] and *abstract regions* [WM04]. ObjectStreams is more versatile, however, as it permits transmission of more complex data structures and multiple communication patterns and protocols.

The combination of functionality provided by ObjectStreams is unique for WSN platforms. The need to provide applications with communication mechanisms larger than or independent of physical packet size is recognized. The *Rime* [Dun07] protocol stack for the Contiki WSN OS includes the *Rudolph* protocol that communicates messages of up to 1024 bytes in size, split into multiple packets. Additionally, Contiki contains the uIP protocol, a minimal implementation of the TCP/IP protocol stack [Dun05], and both TinyOS and Contiki have implemented the 6LowPAN IPv6 protocol [HC08]. Both Rudolph and TCP do not provide the level of functionality that ObjectStreams does, however. First, TCP is intended for a symmetric bidirectional connection, whereas wireless sensor networks employ mostly asymmetric connections such as broadcasting or tree-collection or dissemination, where TCP cannot be used. Furthermore, the Rudolph protocol uses only a single transmission and reception buffer on a sensor node, which means that only a single multi-packet transmission can be in progress at any one time. For applications such as the intruder detection scenario which uses a neighborhood gossip protocol, this is not a suitable solution. In contrast, ObjectStreams is able to simultaneously receive multiple multi-packet messages.

To address the limitations of TCP in the context of wireless sensor networks, transport protocols have been developed [WDLS06]. Some of these address the issue of reliably sending a stream of packets an perform retransmissions when necessary. Rather than being an alternative to ObjectStreams, these protocols are complementary: These protocols still provide a packet interface to applications whereas ObjectStreams translates application-level values into sequences of packets. Furthermore, transport protocols concern themselves with congestion control and retransmissions, something not covered by ObjectStreams.

The second part of ObjectStreams' functionality is communication of language-level data structures. We use a form of object serialization, a technique common for so-called *managed* platforms, such as Java, or Python where this is called *pickling*. Such platforms use serialization or picking when objects need to be stored on disk files or communicated. Within the WSN platforms only the Perk [Cor08] Java platform for WSNs uses a similar communication mechanism. It uses Java's serialization to communicate.

ObjectStreams combines both its parts – multi-packet messaging and serialization – into a combined whole, optimized specifically to efficiently communicate SensorScheme programs and application data.

Chapter 8

Macro-programming by program specialization

This chapter describes an extension to the SensorScheme platform that enables macro-programming a network of heterogeneous WSN nodes using a technique called program specialization. We will demonstrate this technique with an implementation of the *smart office* application scenario.

We start this chapter with a re-iteration of the smart office scenario (see Section 3.4), and define it more precisely, after which we outline a solution and further describe the SensorScheme extension that is the subject of this chapter.

8.1 Smart office scenario

The smart office scenario uses electrical appliances containing wireless sensors and actuators such as light switches, radiator valves and infrared presence detectors. Together, these devices make up a wireless sensor and actuator network used to control both manually and automatically the devices in every office room. Our main goal is to devise a method to administer this network efficiently in the face of changes to the devices' configuration.

A building manager places devices of different types on one of the building's floors using a floor plan as shown in Figure 8.1. A database stores the devices – their location and types – according to the schematic entity relation diagram in Figure 8.2. Additionally, the database contains descriptions of all rooms in

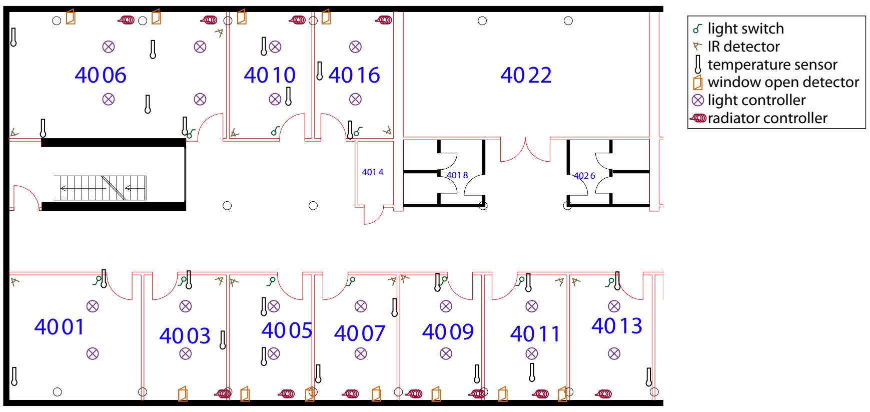


Figure 8.1: Floor plan for use in the scenario

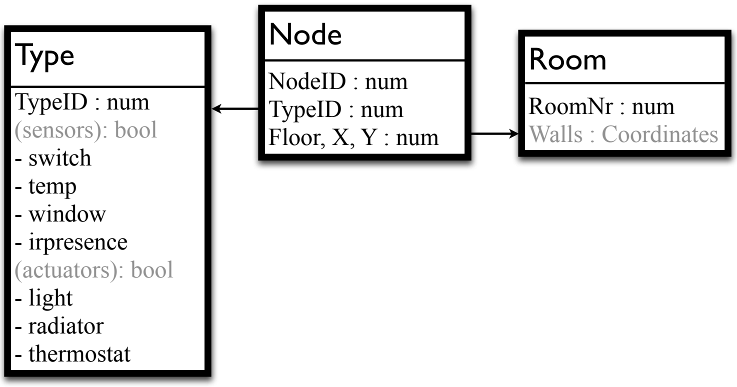


Figure 8.2: Entity Relationship diagram for database

```
(ssmodule office-constants
  (require "thesis-base.ss")
  (provide types-tbl nodes-tbl rooms-tbl)
  ; Types alist with key TypeID and values
  ; (sensors):  switch temp window irpresence
  ; (actuators):  light radiator thermostat
  (define-const types-tbl
    '([1 . (#f #t #f #f #f #f #f)]
      [2 . (#f #f #f #f #f #t #f)]
      [3 . (#t #t #f #f #f #f #f)]
      [4 . (#f #f #f #f #t #f #f)]
      ))

  ; nodes alist with key nodeID and values:
  ; typeID coordinates (floor X Y)
  (define-const nodes-tbl
    '([10 . (1 (4 7 11))]
      [15 . (2 (4 9 12))]
      [93 . (3 (5 8 1))]
      [45 . (4 (5 10 6))]
      ))

  ; rooms alist.  columns :  room nr, bounding coordinates of room
  (define-const rooms-tbl
    '([4006 . (4 (0 9) (8 14))]
      [4010 . (4 (8 9) (11 14))]
      [5010 . (5 (8 9) (11 14))]
      ))
)
```

Listing 8.1: Constants derived from database

the building by its bounding coordinates.

A single generic program, shown in Listing B.5, controls the operation of all devices in the network. This network-wide generic program is to be compiled together with the *constant* declarations in Listing 8.1 obtained from the configuration database. The generic program contains a number of auxiliary functions followed by conditional definitions, one for each kind of sensor or actuator. In the program fragment, the definitions specific to temperature sensors and radiator controllers are shown.

From this generic program, each device receives a specialized variant of this application, depending on its unique node identifier '**id**'. We call this process *specialization*. For a radiator controller device (for example node no. 15) specialization yields the following program (identical to Listing B.5 lines 83-84):

```
(define-handler (report-temp-hdl val)
  (adjust-temp (- (room-temp 15) val))))
```

Using the data obtained from the database it is possible to statically determine that the node is a radiator controller (i.e., `(node-is-radiator id)` must return a `#t` value), and that it is not a temperature sensor (`(node-is-temp? id)` must return `#f`). The specialized program consists of only the two-line handler definition `report-temp`.

Similarly, for temperature sensor device no. 10, specialization results in this program (derived from Listing B.5 lines 61-79):

```
(define collect-ls ())
(define-handler (collect-temp-hdl rn val)
  (if (= rn 4006)
     (set! collect-ls (cons val collect-ls))))

(define (sense-temp-loop t)
  (call-at-time (+ t (* 16 60)) sense-temp-loop)
  (when (= 10 (max collect-ls))
    (send 22 (msg report-temp-hdl (average (map rest collect-ls))))
    (send 23 (msg report-temp-hdl (average (map rest collect-ls)))))
  (set! collect-ls ())
  (bcast (msg collect-temp-hdl 4006 (temp-sensor))))

(sense-temp-loop (now))
```

As expressed in this program, temperature devices in the same room periodically sense and broadcast the temperature (in a `collect-temp` message), which they collect into a list – `collect-ls`. One of the devices in each room (the one with the highest `id`) averages the measurements, and reports the result to all radiators in the room, which are numbers 22 and 23 in this case.

8.1.1 Effects of specialization

The two examples above show some of the possible effects of specialization:

- Some expressions are replaced with their results if the results are *known* – that is, can be proven to always evaluate to the same value. The call to function `room-of` on line 63 of Listing B.5 can be statically determined given the configuration constants declared in module `office-constants` (Listing 8.1) and the function definition.
- Conditional expressions around definitions ensure that only those definitions that are used by a device will be included in its specialized program. Note that the conditional definitions use a normal `if` expression, in contrast to for example the use of the #IFDEF macro in C/C++. In these cases the conditional statement's predicate expression must be a *known* expression.

- The list traversal and subsequent conditional statement in lines 71-74 of Listing B.5 specialize to multiple (two in our example) expressions. Expression `for-each` iterates over the result of the *known* function `nodes-in-room`, which can be determined from the configuration constants (with obvious result). The list of controlled devices is known statically during specialization, and the `for-each` expression can be replaced by concrete calls to `send` to each of those devices.

- The constant definitions from the configuration database are removed from the specialized program, as well as accessor functions and other definitions not referred to in the specialized program. Omission of unreferenced program parts is a major cause of the size reduction that specialization can achieve.

8.2 Partial evaluation

The program specialization described in this chapter is implemented as an extension to the SensorScheme platform, and works by applying *partial evaluation*.

According to Jones *et al.* [JGS93] partial evaluation is a program transformation which optimizes a program with respect to its invariant input data. To our partial evaluator, the invariant input data consists of constants defined in the program and literal values, including a device's unique **`id`**.

In contrast to most uses of partial evaluation, where it is used to speed up the running time of an application, the goal for our partial evaluator is to produce short programs that take up little memory and can be transmitted to the target node efficiently.

Our partial evaluator takes a generic, network-wide SensorScheme program, and specializes it for a node. The result of is a SensorScheme program specialized for a single node, that is subsequently transferred and deployed to the intended node. The process may be repeated to specialize a program for all nodes in the network.

The method described here makes some changes in the *analyzer* process of the SensorScheme tool chain described in Section 6.2.1. In addition to *finding used definitions* the compiler will specialize the definitions in a program with respect to the node identifier to which the program belongs. The specialization process transforms the *initialization expressions* and recursively all global definitions. During specialization, new definitions may be created.

The specialization extension introduces a new primitive expression **`define-const`** to the SensorScheme interpreter definition in Section 6.5.1, which extends the global environment with named constants: immutable values that may be used by the partial evaluator to specialize a program.

8.2.1 Definition

The program specializer described in this chapter extends the functionality of the *analyzer* process described in Section 6.2.1. It syntactically transforms the procedure definitions in a source program and all its required library modules into more specialized procedures where possible, before performing the step of finding all definitions used in the specialized procedures.

The program structure of the specializer mirrors the evaluator defined in chapter 6, consisting of a specialization function **SE** analogous to `eval` (or **E**) in Listing A and a function **SA** specializing applications, analogous to `apply` (or **A**). The difference is that whereas functions **E** and **A** evaluate an expression with the purpose to obtain a result value and perform side effects, functions **SE** and **SA** merely transform an expression into an equivalent but simpler one that, when evaluated, will perform the same operations and produce the same results as the original expression. This difference is expressed in the notations of **E** and **SE**: where **E** may perform multiple actions separated by semicolons and produce a result (with 'return'), **SE** expresses its transformation through the operator $\Longrightarrow$, analogous to the definition of macro's in Section 5.2.10.

Variables and environment

$$\mathbf{G} = \mathbf{G}_{init} \cup \{\mathbf{id} \mapsto id\}$$

During the transformation process, the specializer has access to a *global environment* **G**, similar to the evaluation process. The global environment **G** is initialized with a set of initial bindings $\mathbf{G}_{init}$ such as primitive procedures, and the device's identifier **`id`**, bound to the device's id number. Programs may use the node's id number to compare it with some constant value, typically as part of the predicate part of an **`if`** expression.

Again, mirroring the evaluation process, specialization function **SE** accepts as argument the local environment ε. Both environments **G** and ε bind variable names to the values these variables will be bound to during evaluation: if the value bound to a variable is a constant known at specialization time, this constant value binds to the variable. Otherwise, when the value bound at run-time is not constant or cannot otherwise be determined by partial evaluation, a variable binds to the special value ⊥.

Literals

```
$\mathbf{SE}(expr,\ \varepsilon)$ when (number? $expr$)  |
                (boolean? $expr$) |
                (null? $expr$)
    $\Longrightarrow$  $expr$

$\mathbf{SE}($(quote $literal$)$,\ \varepsilon)$
    $\Longrightarrow$  (quote $literal$)
```

Just like evaluation function **E**, **SE** operates depending on the type and structure of its input expression *expr*. In the simplest case, when *expr* is a literal value, no transformation needs to take place, and **SE** produces the original expression *expr*.

Variable references

```
$\mathbf{SE}(expr,\ \varepsilon)$ when (symbol? $expr$)
      let  $v = \varepsilon[expr]$     if  $expr \in \varepsilon$
               $\mathbf{G}[expr]$  if  $expr \in \mathbf{G}$
               error      otherwise
    $\Longrightarrow$  $v$      if (constant? $v$)
          $expr$  otherwise
```

Variable references may specialize to produce the value bound to the referred variable. For a variable that is bound to a known constant value, a variable reference transforms into the value the variable is bound to. Otherwise, if a variable is bound to an unknown value ⊥, no transformation takes place.

Conditionals

```
$\mathbf{SE}($(if $pred\ conseq\ alt$)$,\ \varepsilon)$
      let  $v = \mathbf{SE}(pred, \varepsilon)$
    $\Longrightarrow$  $\mathbf{SE}(alt, \varepsilon)$                                     if  $v$ = #f
          $\mathbf{SE}(conseq, \varepsilon)$                                  if  (constant? $v$)
          (if $\mathbf{SE}(pred, \varepsilon)$ $\mathbf{SE}(conseq, \varepsilon)$ $\mathbf{SE}(alt, \varepsilon)$) otherwise
```

Conditionals specialize to either the *consequent* or *alternative* expression if the *predicate* can be reduced to a constant. If not, a complete conditional expression is returned, in which its subexpressions may be more specialized.

Definitions

```
SE((define var value), ε)
     G ← G ∪ {var ↦ ⊥} ;
   ⟹ (define var value)

SE((define-const var value), ε)
     G ← G ∪ {var ↦ SE(value, ε)} ;
   ⟹ (define-const var value)
```

Top-level variable and constant definitions create new bindings in **G**. A non-constant definition `define` creates a binding that is mutable at runtime, and binds to $\perp$ during the specialization process. A constant definition `define-const` creates a binding to a constant value that can be retrieved during specialization with a variable reference.

Assignments

```
SE((set! var value), ε)
   ⟹ (set! var SE(value, ε))
```

Reassignment of both local and global variables represents a side-effect and must occur at runtime. Specialization does not transform the `set!` primitive expression.

Lambda expressions

```
SE((lambda (var_1 ... var_n) expr_1 ... expr_m), ε)
     let  ε* = ε ∪ {var_1 ↦⊥} ... {var_n ↦⊥}
   ⟹ (proc ε (var_1 ... var_n) SE(expr_1, ε*) ... SE(expr_m, ε*))
```

Specialization of a `lambda` expression produces a procedure, which captures the current local environment ε, and specializes the lambda expression's body expressions using an environment extended with the procedure variables $var_1 \ldots var_n$ bound to $\perp$. When this lambda expression is used in an application, its body expressions will be specialized again according to the application rules defined below.

```
SE((proc ε_c (var_1 ... var_n) expr_1 ... expr_m), ε)
   ⟹ SE((lambda (var_1 ... var_n) expr_1 ... expr_m), ε)
```

In the case of nested lambda expressions, for example as a result of the use of `let` expressions, the specialization rules may cause the inner lambda expression to be evaluated multiple times. During specialization, procedure expressions are treated as a re-evaluation of a lambda expression. Re-evaluation may result in a more specialized procedure if the environment contains additional constant values rather than bindings to ⊥, as a result of any of the application roles following below.

Procedure calls

```
SE((fn arg_1 ... arg_n), ε)
    ⟹ SA(SE(fn, ε), SE(arg_1, ε) ... SE(arg_n, ε))
```

If the expression to specialize is not any of the previous primitive expressions, the expression is an application. Analogous to the execution of applications, function **SA** specializes the application of a function fn on its arguments arg_1 ... arg_n.

Apply

Function **SA** distinguishes a number of following cases outlined below:

Primitive application

```
SA(prim, arg_1 ... arg_n) when (primitive? prim)
    ⟹ prim(arg_1, ..., arg_n) if (constant? arg_1) & ... & (constant? arg_n)
       (prim arg_1 ... arg_n) otherwise
```

Primitive applications may be specialized to a single constant. When all of the arguments to the application have been specialized to a constant value, the primitive is applied directly to the arguments, producing a constant value. Otherwise, the expression is not specialized further.

Call elimination

```
SA((proc ε_c (var_1 ... var_n) expr_1 ... expr_m), arg_1 ... arg_n)
              when (constant? arg_1) &...& (constant? arg_n)
     let  ε*_c = ε_c ∪ {var_1 ↦ arg_1} ... {var_n ↦ arg_n}
    ⟹    SE(expr_m, ε*_c)  if (constant? SE(expr_1, ε*_c)) &...&
                              (constant? SE(expr_m, ε*_c))
```

Similarly, procedure calls may be simplified to a single constant when the result of an application is fully known. When the arguments to a call as well as its body expressions all evaluate to known values, the application reduces to only the constant return value of the last body expression.

Lambda application

```
$\mathbf{SA}$((proc $\varepsilon_c$ ($var_1$ ... $var_n$) $expr_1$ ... $expr_m$), $arg_1$ ... $arg_n$)
    let  $\varepsilon_c^* = \varepsilon_c \cup \{var_i \mapsto arg_i\} \ldots,$  if (constant? $arg_i$)
    let  $var_1^*$ ... $var_k^*$, $arg_1^*$ ... $arg_k^* = non-constant\ vars\ and\ args$

    $\Longrightarrow$ ((proc $\varepsilon_c$ ($var_1^*$ ... $var_k^*$) $\mathbf{SE}(expr_1, \varepsilon_c^*) \ldots \mathbf{SE}(expr_m, \varepsilon_c^*)$) $arg_1^*$ ... $arg_k^*$)
```

Application of anonymous functions reduces to an application with possibly fewer arguments $k, k \leq n$ of a more anonymous function, having parameters $var_1^* \ldots var_k^*$. If some argument in the application is known to be constant, it is included in the local environment ε_c^* and removed as an argument, along with its corresponding procedure parameter. Other, non-constant arguments, and their corresponding variables are included in the remaining function.

Constant procedure application

```
$\mathbf{SA}(fn,$ $arg_1$ ... $arg_n$) where $\mathbf{G}[fn] =$ (proc $\varepsilon_c$ ($var_1$ ... $var_n$) $expr_1$ ... $expr_m$)
    let  $\varepsilon_c^* = \varepsilon_c \cup \{var_i \mapsto arg_i\} \ldots,$  if (constant? $arg_i$)
    let  $var_1^*$ ... $var_k^*$, $arg_1^*$ ... $arg_k^* = non-constant\ vars\ and\ args$
    let  $fn^* = fn\ annotated\ with\ \{var_i \mapsto arg_i\}$ ...

    $\mathbf{G} \leftarrow \mathbf{G} \cup \{fn^* \mapsto$(proc $\varepsilon_c$ ($var_1^*$ ... $var_k^*$) $\mathbf{SE}(expr_1, \varepsilon_c^*) \ldots \mathbf{SE}(expr_m, \varepsilon_c^*)$)$\}$ ;
  $\Longrightarrow$ ($fn^*$ $arg_1^*$ ... $arg_k^*$)
```

Similarly, application of procedures bound in $\mathbf{G}$ to variable fn specializes to an application with possibly fewer arguments. Specialization returns an expression applying its arguments to a new variable, formed from the original variable fn, annotated with bindings of those variables that receive constant arguments, to their constant values. This new variable is bound in $\mathbf{G}$ to a procedure obtained from the procedure bound to fn with possibly fewer free variables.

8.2.2 Mutability

SensorScheme makes use of the `set!` primitive expression to modify variables both in the local and global environment. To handle assignments properly, the specializer engages in some code analysis during specialization not represented in the definition above: when specializing lambda or constant procedure applications all procedure parameters are checked for the possible occurrence of assignments involving any of its parameters. Any variable that is assigned to binds to $\perp$ instead of possibly the known value of its corresponding application

```
(define-const nodes-in-room (quote (9 17 22 23 30)))

(define-const radiatorNodes (quote (22 23 35)))

(define-const sense-temp-loop
  (lambda (t)
    (call-at-time (+ t (* 16 60)) sense-temp-loop)
    (for-each (lambda (n)
                (if (member n radiatorNodes)
                    (send n (msg report-temp ...))))
              nodes-in-room)))

; library definitions
(define-const for-each
  (lambda (fun list)
    (if (not (null? list))
        (begin (fun (car list)) (for-each fun (cdr list))))))

(define-const member
  (lambda (x list)
    (if (null? list) #f
        (if (eq? (car list) x) #t (member x (cdr list))))))
```

Listing 8.2: Fragment of example scenario program

argument.

Data structures are considered not mutable to our specializer. The only data structures SensorScheme uses (besides procedures) are *pairs*. Primitive operations `set-car!` and `set-cdr!` are not permitted in the context of the specializer.

For top-level variables bound in **G**, its occurrence in a **`set!`** expression is illegal if the variable was defined as a constant (using **`define-const`**). Top-level variables – defined using **`define`** never bind to known values, and therefore are never subject to specialization.

Top-level function definitions must be declared as constants using **`define-const`**. **`lambda`** expressions are considered constant values as well, and are therefore allowed value bound to top-level constants. Although variable function definitions using **`define`** are allowed, they will not be subject to specialization.

8.3 Operation

In this section we will again use the smart office application scenario to show the operation of the specialization method.

Listing 8.2 shows a fragment of the program given in Listing B.5. The function `sense-temp-loop` is part of the program to be executed by temperature sensors. We have made it somewhat simpler (replacing a call to `nodeInRoom` with a constant table with the result of the function, and a call to `node-is-radiator` replaced by a membership check of table `radiatorNodes`), but it is identical in intent.

While processing this program the specializer encounters the following fragment

```
(member n radiatorNodes)
```

and start specializing it by replacing the constant references (and specializing those) like this:

```
((proc {t,n} (x list)
   (if (null? list) #f
       (if (eq? (car list) x)
           list
           (member x (cdr list)))))
 n (quote (22 23 35)))
```

We have written the local environment as a list of variables within brackets. Next is specialization of the only body expression of the `proc` expression, using the extended environment `{t, n, x, list=(quote (22 23 35))}`. The `if` expression's predicate is a primitive call yielding `#f`, causing only the *alternative* part to be specialized. The complete specialization, after evaluation of the primitive calls `(car list)` and `(cdr list)` results in this specialized expression:

```
((proc {t,n} (x)
  (if (eq? 22 x) #t
      (if (eq? 23 x) #t
          (if (eq? 35 x) #t
              #f))) n))
```

This call was created from a call to a globally defined constant `member`, specialized on its `list` parameter. So instead, the specializer creates a global function, annotated with the specialized parameters and its values, and with just one parameter left:

```
(define-const member<list=(quote (22, 23, 35))>
  (proc {t,n} (x) ... ))
```

and replace the original call with:

```
(member<list=(quote (22, 23, 35))> n)
```

In fact, each call to `member` specializes to a call to a newly created specialized function. We have in-lined these functions here for readability.

After specializing only a small part of our program, we can already see some of the consequences of specialization. Even though our goal was to minimize program sizes, specialization of the member function results in a longer function than the original. If we were to use a longer radiatorNodes list, the specialized function would increase in size proportionally. The program does, however, not depend anymore on radiatorNodes, since all its information is now encapsulated in the specialized member function.

Specializing the sense-temp-loop proceeds similarly. The recursive calls to for-each create new versions specialized to the tails of nodes-in-room. For each item in the list, this produces a call to

```
(proc {t} (n)
  (if (member<...> n)
      (send n (msg report-temp ...))))
```

with the list item as argument. For each of these the call to member<...> resolves to either #t or #f, including resp. excluding the send call from the final function. The final result, as stated earlier is the following:

```
(define-const sense-temp-loop
  (lambda (t)
    (send 22 (msg report-temp ...))
    (send 23 (msg report-temp ...))))
```

8.3.1 Post-processing

In fact, this is not exactly the code specialization produces. After partial evaluation of the program, we employ a post-processing step to produce the resulting program presented above. Post-processing is responsible for a number of transformations to reduce the size of the final program:

To start with, proc expressions are transformed into regular **lambda** expressions that my appear in SensorScheme programs.

Next, specialization of top-level functions produces new, annotated and specialized versions of the original function. Many of these are referred to in a call only once or even not at all in the final program. Those referred once are in-lined at their call site, and those unreferenced removed from the final program.

Finally, some trivial simplifications: ((**lambda** () a ((**lambda** () b c)))) transforms to ((**lambda** () a b c)).

8.3.2 Program size

This chapter has shown how partial evaluation as a method of program specialization can be used to generate short, device-specific programs. Partial evaluation, however, is known to cause significant growth of program sizes, a fact we can also observe from the description above: 'loop unrolling' of recursive functions (our `member` and `for-each` functions) specialized witch respect to long lists may create excessively long programs, far longer than the original, unspecialized one. The specialized `member` functions are, however, not part of the final program, and therefore have no impact on the total program size.

The particular setting in which we use the technique does result in small programs. Even though networks may be very large, up to hundreds of devices, every individual device interacts with only a small number of others. The configuration tables may grow very large, and intermediate specialization functions proportionally so. When carefully constructed, the final program is proportional in size to only the peers a device interacts with directly.

8.3.3 Discussion

The goal of writing a single network-wide program for wireless sensor networks to better facilitate application development and management has received much research attention already (see Section 2.6.3) Its aim is to allow users to more easily deal with the inherently distributed nature of WSN applications by lifting communication and concurrency operations to the language level. These systems are, however, intended for homogenous wireless sensor networks, where all nodes perform the same task and contain the same program.

Our work has most in common with the snBench [BBKO05] RuleCaster [BK06] ATaG [BPRL05] and Titan [LRST07] systems. In all of these, applications are modeled as 'task graphs' where graph nodes represent tasks or computations, and edges represent the flow of data between tasks. The goal then is to subdivide a network-wide task graph into sub-graphs which can each be placed on different devices. The edges connecting the distributed sub-graphs transport their data transparently over the wireless network. The major challenge for these systems is to find a division of the application graph resulting in low energy consumption and high communication reliability.

While some similarities are immediately apparent (automatic division of single program into node-specific smaller ones; the use of graph or tree structures to represent and transform programs), a number of important differences exist as well: the program structure of interconnected tasks is more limited than the Scheme platform we use: no globally mutable state accessible by multiple tasks exists, and the graphs are acyclic in nature, disallowing feedback loops and other program control flow.

Another major difference is that where task graphs are split implicit network communication is added to the deployed application. Because communication is one of the major consumers of scarce energy, the particular division strategy has high impact on the network's performance and life time. Our method does not introduce any implicit communication into the program.

Probably the most prominent difference is that our method explicitly takes into account the difficulty around management of large networks, where many nodes perform somewhat similar tasks. Our method provides a straightforward way to separate the general functionality from the node-specific configuration.

Partial evaluation for Scheme-like systems is a well-researched topic. In fact, Scheme is the language of choice for some of the most extensive work on partial evaluation, such as Jones et al.'s [JGS93] comprehensive description of the topic. In fact, the most powerful and complete partial evaluators such as MIX [JSS89], Similix [Bon91], Schism [Con93] and Fuse [Ruf93] use some form of Scheme as both their implementation and transformation language.

The use of the SensorScheme platform requires our partial evaluator to work in the presence of mutable variables. Most research on partial evaluation employing Scheme-like languages operates on a subset of Scheme or an extended λ-calculus instead of a practically used language, as the above-mentioned systems do. To accommodate partial evaluation, SensorScheme is modified by adding global constant definitions and allowing only immutable pairs.

We make use of an *online* partial evaluator with *polyvariant specialization* according to Ruf's definitions [Ruf93] and employ many techniques described in the same work. The major difference (and technical novelty) to Ruf's online partial evaluator is our method of specializing and annotating global function calls.

Our use of partial evaluation – namely to reduce program size, instead of focusing on increasing speed of computation – is novel (to the best of our knowledge), and may seem somewhat counter-intuitive. A well-known effect of partial evaluation is explosion of program size. The typical use of partial evaluation in the application context does make this unlikely to happen. The use of partial evaluation we discussed in this chapter aims to fragment a program and a – possibly large – set of constants into a small specialized program.

Chapter 9

Evaluation

This dissertation presents a wireless sensor network platform. We presented a broad goal for this platform by describing the characteristics and limitations of the hardware used (Chapters 2 and 4), and its intended functionality by way of a number of example application scenarios to be implemented on this platform (Chapter 3). We subsequently proposed a platform design (Chapters 6–8), and presented implementations for these application scenarios (Chapter 5). This chapter evaluates how the SensorScheme platform meets the stated goals by measuring performance characteristics of the example implementations and comparing them to the state of the art.

9.1 Computation and energy efficiency

Wireless sensor network platforms have limited computational resources to meet the demands of cost and form factor as we described in Chapter 2. Sensor-Scheme incorporates design decisions that potentially consume more resources than otherwise needed. This section determines whether SensorScheme is able to meet the tight resource budgets of WSN platforms, considering the following measures of efficiency:

1) Interpretation causes a slowdown in program execution compared to executing native processor instructions. To what extent do interpreted Sensor-Scheme programs cause significant delay in program execution, and consume significant energy during the longer program execution?

2) Low power sensor nodes contain only little memory. Memory consumption

of SensorScheme programs should stay within the limits of the available memory. How well is the SensorScheme platform is able to meet the demands

The following sections present measurements performed with the SensorScheme implementation described in Chapters 6 and 7. We have not included experiments that directly compare SensorScheme's performance against other WSN platforms, for a number of reasons. First, the TinyDB and Maté platforms have been written for older releases of TinyOS version 1, and it has proved impossible to execute these applications reliably on WSN devices due to software bugs. Performing experiments to compare their performance to SensorScheme's has therefor not been possible.

Second, many of the platforms and applications comparable to SensorScheme were written for different hardware platforms, with different CPU characteristics (8-bit vs. 16 bit, Harvard vs. von Neumann architecture) and wireless networking standards (See Table 9.5). With the current state of WSN hardware and software platforms, porting an application to run on a different platform than it was written for is a complicated and time-consuming task, as applications are often written with the hardware characteristics of the platform in mind.

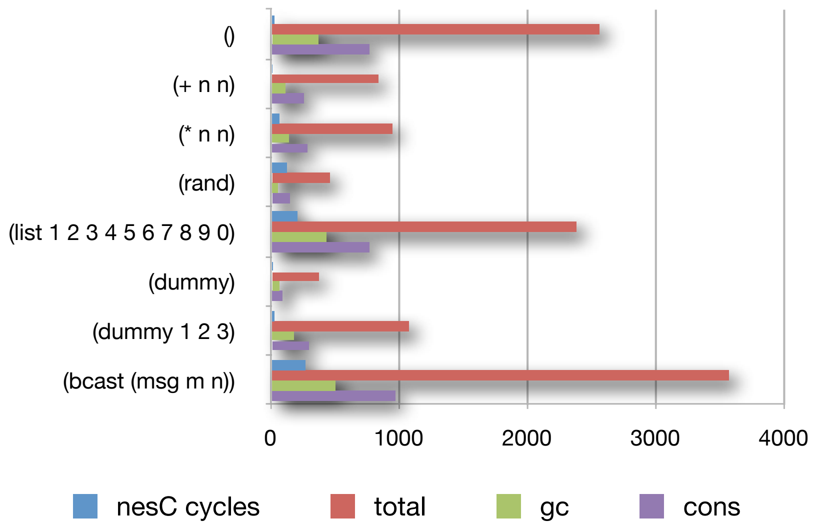
But most importantly, the different platforms and applications each have different characteristics and limits to the programs they can execute. This makes it impossible to write the 'same' program for each platform. For example, TinyDB supports a more limited range of queries than SSQuery or SwissQM. Furthermore, these platforms use different algorithms, communication protocols and data structures to implement these queries. Direct comparisons of this kind do not gain additional insight into individual performance characteristics. We will address this issue more closely in Section 9.3.

9.1.1 Execution performance

The interpreted nature of SensorScheme imposes an execution overhead in comparison to native code, which causes increased energy use on wireless sensor nodes. To quantify this overhead we have performed two measurements: a micro-benchmark to quantify the increased execution time of interpreted SensorScheme programs, and an application test run to measure the energy spent for the execution of SensorScheme programs.

Microbenchmark

We have measured the computation time of a number of simple test cases in SensorScheme, by repeatedly executing them in a tight loop, and compared it



		SensorScheme			nesC	
		Total cycles	% gc.	% cons	cycles	fraction
1	`()`	2557	14%	30%	25	102:1
2	`(+ n n)`	837	13%	30%	8	105:1
3	`(* n n)`	945	15%	30%	64	14.8:1
4	`(rand)`	455	12%	32%	123	3.70:1
5	`(list ...)`	2377	18%	32%	204	11.7:1
6	`(dummy)`	370	17%	24%	13	28.5:1
7	`(dummy 1 2 3)`	1072	17%	27%	26	41.2:1
8	`(bcast ...)`	3565	14%	27%	267	13.4:1

Table 9.1: Results of interpretation overhead measurements

to native execution speed of the same operations programmed in nesC.

All tests were performed using a MSP430 processor emulator that accurately counts the clock cycles per instruction. Table 9.1 shows the results of these measurements.

For every test we have measured the total number of instructions to execute the test 1000 times in a loop for both the SensorScheme and nesC programs. From this we calculate the average number of cycles spent executing each test once, shown as 'total' resp. 'nesC cycles' in Table 9.1. We furthermore measured the number of instructions spent in SensorScheme for allocating memory cells ('cons') and the cycles spent executing the garbage collector ('gc').

- `()` : The first test case measures the running time of the loop itself, representing a simple case of flow control.
- `(+ n n)`, `(* n n)` : Cases two and three perform simple (addition) and more complex (multiplication) arithmetic operations.
- `(rand)` : Case four calculates a more complex native procedure, random number generation.
- `(list 1 2 3 4 5 6 7 8 9 10)` : Case five tests dynamic memory allocation. It tests a call to `malloc` and `free` in the nesC program.
- `(dummy)`, `(dummy 1 2 3)` : Cases six and seven evaluate the overhead of function calls resp. without and with parameters.
- `(bcast 1 2 3 4 5 6 7 8 9 10)` : The last test case measures the cost of communication.

The results show that more complex operations, such as executing long sequences of native code (cases 3 and 4) impose less overhead than simple arithmetic operations and control flow (cases 1 and 2). Furthermore the results suggest that SensorScheme's uniform memory lay-out results in comparatively cheap memory allocation (case 5) and that function application is a relatively inexpensive operation compared to other flow control (cases 6 and 7).

All test cases spend a similarly large fraction of time for garbage collection and memory allocation – approximately 15% and 30%. SensorScheme's evaluation process involves allocation of cells for the call stack and environment, incurring a significant runtime cost. This quickly consumes all memory, after which a garbage collection cycle is necessary, which again frees up most of the allocated cells.

These results are not unlike those reported for other virtual machines. For example, Maté reports execution slowdown of 3.4 - 33 $\times$ for operations such as arithmetic and I/O operations. Interpreted languages for desktop systems have similar performance characteristics. The online benchmark "The Computer Language Benchmarks Game" [sou09] shows median execution speed ratios of C vs. some interpreted languages of 13:1 (PLT Scheme : C) to 78:1 (Ruby : C) [1].

Application performance and energy use

The next experiment uses the two-hop intruder detection application shown in Listing 5.5 to obtain an insight in the computation time and energy use of a real application.

The test uses a TOSSIM simulation network of 20 nodes and an additional node that is simulated in our MSP430 processor emulator also used in the previous test. In this test we measure the computation time and number of messages sent and received to calculate the energy spent in a single *period* – the time between subsequent calls to `time-loop`. All time and energy calculations assume running this program on a network of Tmote devices (see table 2.1.6) at 8 MHz. Power consumption data is based on the data sheets of the hardware components.

Table 9.1.1 (a) lists some results of the per period running time. For each such period, the SensorScheme code takes only 156 ms execution time. With a period of 5 seconds this is just three percent of the period duration. While we may assume that the same program written in nesC will execute in far shorter time, SensorScheme's slower execution time does not slow down the application as a whole.

Similar to the micro-benchmark test, a large fraction (about 57%) of execution time is spent on memory allocation and garbage collection. Garbage collection itself causes application pauses of only 10 ms, an acceptable delay for most WSN applications.

Communication takes a significant fraction of the total energy use on WSN nodes. Table 9.1.1 (b) shows the number of messages sent and received per period, and the energy spent on computation by the OS, based on estimations, and the energy use of the radio during sending and receiving. (Before sending, the radio needs to power up taking an additional 3 ms, included in the air time.)

[1]The single core x86 benchmark of October 12 2009: `http://shootout.alioth.debian.org/u32/benchmark.php?test=all&lang=all&lang2=gcc&box=1`

(a)		cycles	ms	mJ
	Execution time and energy			
	TinyOS	156636	20	0.16
	SensorScheme	1245483	156	1.27
	Fraction spent in allocation	25.2%		
	Fraction spent in GC	31.4%		
	– # collections	6.43		
	– execution time / collection	7.6	ms	
	– avg. used cells	395	cells	
	– max. used cells	429	cells	

(b)	Comm. energy	TX	RX	total	
	`neigh-msg`	2.5	11.6		msgs
	message size	324	352		bits / msg
	air time	11	51	62	ms
	radio energy	0.29	1.96	2.25	mJ

(c)	Total energy used			
	TinyOS execution	0.16	mJ	4.3%
	SensorScheme execution	1.27	mJ	34.5%
	Radio TX / RX	2.25	mJ	61.1%
	Total	3.68	mJ	

Table 9.2: Execution statistics of intruder detection application

Finally, taking these three sources of energy use together, table 9.1.1 (c) shows the relative cost of each of those. It shows that most energy is used by the radio power during communication (61 %), while SensorScheme execution time takes 34.5 % of the total energy spent. The same program written in nesC may result in an energy reduction of no more than 34.5%.

We have not taken into account other sources of energy use which are independent of the execution platform, such as MAC protocol overhead (idle listening) and sensor readouts, which only reduce the fraction of energy used by program interpretation.

9.1.2 Conclusion

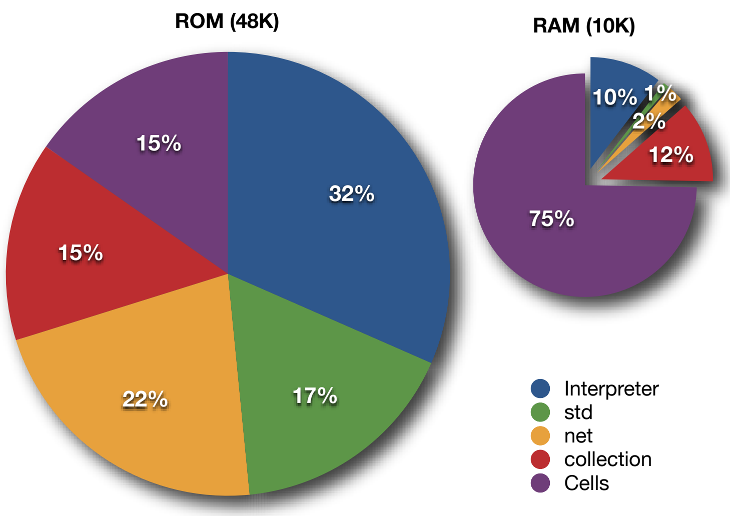
SensorScheme performs well enough to be used in real WSN scenarios: increased energy consumption due to interpretation is only moderate. Depending on the application context, this energy consumption increase may well be a price worth paying to benefit from Sesorscheme's architectural characteristics. Moreover, the current SensorScheme implementation is a prototype which may be improved in several ways to gain a better execution or energy performance. As an example, a significant portion of execution time is spent on memory management activities, both allocation and garbage collection. Future versions may reduce this cost and make SensorScheme more energy efficient.

9.2 Program size and complexity

Besides the computation and energy performance described above, a crucial metric for WSNs are the size of applications and the SensorScheme interpreter. SensorScheme application size is relevant as this determines the energy cost and duration of reprogramming nodes on one hand, and because it occupies the very limited available memory of sensor nodes.

Code complexity

We start this section by taking a closer look at the environmental monitoring scenario. This is a frequently-used scenario that has been the focus of a number of platforms described in the state of the art. We compare a number of implementations that use a distributed query to express the data to be obtained from the network as we described in the environmental monitoring scenario in Section 3.2:



configuration	bytes in ROM	bytes in RAM
Interpreter	15522	1066
std	8286	90
net	10680	228
collection	7150	1206
Cells	(1878 cells) 7514	(1912 cells) 7650
Total	49152	10240

Table 9.3: Thesis-base memory use with different sets of primitives

Application	external repr. (bytes)	network repr. (bytes)	internal repr. (cells)
sensor-blink	337	41	39
eval	3209	575	587
intruder-single	2410	318	328
intruder-twohop	2503	396	411
monitoring	1147	186	194
ssquery	3474	306	303

Table 9.4: Program sizes of the application scenario implementations in each of the three used program representations.

TinyDB
: [MFHW03], an early implementation of this scenario;

QueryVM
: [LGC04], using the Maté virtual machine to execute queries;

SwissQM
: [MRD+07] A more recent platform using a specially-developed virtual machine to evaluate event monitoring queries.

SSQuery
: SensorScheme's implementation discussed in Section 5.8.

All these applications perform roughly the same function, albeit using quite different strategies. We compare the sizes of source code and compiled binary images, which are an indicator of the complexity of the applications. While code size is not a very accurate indicator for the complexity of a program, it gives a coarse indication of the complexity of the SensorScheme implementation compared to other, similar applications.

Table 9.5 shows the measurements for these applications. The measured applications are built for a number of different platforms, and for different versions of TinyOS, as shown in the second column of the table. This makes the data more difficult to compare. The third column shows the number of lines of code in the "app.c" file produced as an intermediate stage of the TinyOS compilation process. It contains all source code that is to be compiled into the binary image (as C code, instead of nesC, which is the source language of the OS and applications), containing the application, library and operating system

Application	Operating system & Platform	app.c lines of code	ROM (bytes)	RAM (bytes)
TinyDB	TinyOS 1, Mica2	19157	60078	2629
QueryVM	TinyOS 1, MicaZ	19778	70730	3639
SwissQM	TinyOS 2, TMote	25812	49110	4679
SSQuery	TinyOS 2, TMote	21990	41994	2590
SSQuery	TinyOS 2, MicaZ	17743	54984	2834

Table 9.5: Binary image size of a number of implementations of the distributed database querying application

code. The last two columns contain the ROM and statically allocated RAM sizes of the compiled binary image.

Data for SSQuery is shown twice, once compiled for the TMote hardware platform and once for MicaZ. Both compile a SensorScheme interpreter with primitives included from the modules `std`, `network` and `collection`, required to run the SSQuery program. We will discuss the SensorScheme configuration in more detail in Section 9.2. These numbers do include the code size of the SSQuery program as an injection message (in network encoding) in ROM, which will be loaded when the devices start up.

The data in Table 9.5 indicates that the SensorScheme implementation is somewhat smaller than the other implementations. SSQuery for the TMote platform uses about 85% of the ROM. Similarly, The SSQuery for the MicaZ platform is about 77 % of the QueryVM implementation and 92 % of the size of TinyDB. Still the SensorScheme implementation is more flexible: The capabilities of SSQuery extend beyond those of any of the other implementations as we have shown in Section 5.8. Furthermore, it is capable of executing not only the SSQuery program, but all of the programs we have discussed in this work.

Configuration and primitives

SensorScheme interpreters compiled into a code binary are configured to contain a selected set of primitive procedures. All applications in Chapter 5 use an interpreter with primitives configured in the base module `thesis-base` (see Appendix C.1). We have compiled a number of different configurations to assess the sizes of the different primitives, shown in Table 9.3. The SensorScheme interpreter and ObjectStreams encode and decoder, without any primitives occupies about 15KB of ROM. Most of the 1 KB or allocated RAM is used for a set of communication buffers. All primitives defined in the standard library, such as operations on base types, and timer, sensor and led access primitives together occupy another 8 KB. The basic communication routines such as `bcast` and the TinyOS communication stack occupy another 10 KB. The *collection* module used by the SSQuery application, containing additional communication primitives based on the TinyOS collection protocol uses an additional 7 KB. All remaining memory in both Flash ROM and RAM, about 7.5 KB each, is available for use as allocation cells, used by SensorScheme applications.

Program size

The last data set in this section, in Table 9.4, presents the program sizes of each of the example programs described in Chapter 5. The program sizes shown are the result of the selection step in Section 6.2.1 and specialization in Section 8.2. At this stage the program contains only those definitions from the program module and any module it `requires` that are referred to in the program. The external representation contains the program as human-readable source code. The network representation is the size of the program as an ObjectStreams message sent into the network to reprogram the sensor nodes. The internal representation contains the number of cells required to store the program in memory. These numbers show that even though SensorScheme applications are compact already in its source format (in external representation) both binary formats are significantly smaller still. Each of these programs can be transported into the network in a few hundred bytes.

9.2.1 Conclusion

This section has evaluated several indicators of program size and complexity of the SensorScheme interpreter and applications written for it.

Our code complexity measurements, comparing SSQuery to other implementations of the environmental monitoring scenario, show some benefits of the SensorScheme design: while the code complexity and memory requirements are lower than other solutions, SensorScheme is able to provide a wider range of functionality. As an example, SSWQuery accepts any user-written aggregation function, instead of just a limited choice of a few functions built-in into the application as does TinyDB. Moreover, the SensorScheme implementation evaluated is a generic interpreter, capable of executing other applications besides

SSQuery, in contrast to the other implementations, which are purpose-built for a WSN distributed query application.

We have furthermore shown the memory use breakdown of the SensorScheme interpreter core and additional sets of primitives (see Table 9.4). The results show that memory consumptions stays within the tight limits of low power WSN nodes, while leaving enough space for memory to be consumed by applications running in the interpreter.

Our program size evaluation shows program sizes of the example SensorScheme programs presented in Chapter 5. Their memory footprint (as internal representation) stays well within the amount available for SensorScheme programs, and the network representation is small enough to transfer a program into the network in just a few seconds, even at the rate of as little as 100 bytes per second or less, as measured with the Deluge protocol [HC04].

Together these measurements show that the SensorScheme design results in an implementation suitable for the memory restrictions of WSN hardware platforms. Moreover, while staying within these bounds, SensorScheme's features and functionality exceed the state of the art, as the SSQuery implementation shows.

9.3 Communication

This section evaluates the performance of ObjectStreams described in Chapter 7. ObjectStreams is itself not a communication protocol but an application-level communication mechanism. Our simulations do not measure communication performance parameters such as throughput or delivery ratio as is commonly the case with evaluations of communication protocols. Instead we compare alternative implementations using the same protocols, and evaluate the effect of different communication abstractions on application performance.

ObjectStreams provides a higher abstraction layer for communication, to easily write communicating applications with few lines of code. A higher abstraction level gives less control over the implementation details, and may affect performance.

Our goal is to evaluate the impact of the use of ObjectStreams to performance parameters such as data reception and energy efficiency. We compare the performance of two of our application scenario implementations – intruder detection (see Section 3.1) and environmental monitoring (see Section 3.2) – to previously published alternatives with a similar abstraction level of communication.

Our evaluations measure:
1) the impact of the communication abstraction on the number of packets sent and received by nodes – a measure of energy efficiency, and
2) the data items received by and memory requirements of the application – a measure of application performance.

We use the TOSSIM wireless sensor network simulator part of TinyOS 2 to measure the performance characteristics. We have modified the network connectivity model to enable us to measure only the applications' effect on packet transmission and reception rates at varying network densities, independent of the physical parameters of the radio hardware such as bit-rate, transmission power, antenna characteristics and environment (indoors vs outdoors).

Before discussing the applications to be evaluated, we first describe the setup of the experiments that follow in Section 9.3.1. We then discuss the evaluations of the two-hop gossip protocol used in the intruder detection scenario in Section 9.3.2 and tree–routed data collection in Section 9.3.3.

9.3.1 Simulation and communication modeling

Wireless sensor networks consist of (a large number of) nodes, collectively performing a common task. Nodes communicate over a wireless interface with a broadcast nature: a data transmission of a single node can be received by multiple other nodes. Not all nodes, however, will be able to receive a transmission, for reasons such as fading and reflection as a result of the distance between nodes, obstacles in-between or nearby the sender and receiver, and interference from other nodes or external sources. As empirical studies show [GKW$^+$02], between any pair of sender and receiver node there is a somewhat fixed probability of message reception, that is related to the distance between the nodes, with a high (but less than 100 %) probability of reception between nearby nodes, and a low probability (but greater than 0) between distant nodes.

Network model

The experiments described below are conducted using the TOSSIM wireless sensor network simulation framework, part of the TinyOS 2 distribution. TOSSIM simulates the simultaneous execution of a single nesC application on a network of devices. The network is modeled as a fully connected graph, with directional edges, labeled with P_{ij}, the probability of reception between any two nodes.

The distance-related distribution parameters are taken from an empirical study performed by Ganesan *et al.* [GKW$^+$02].

We evaluate our protocols using a simulated network of nodes that run the algorithms described below. The network consists of a large number of nodes, placed randomly on a square area of varying size. We obtain the packet reception probability P_{ij} between each pair of nodes from the distance-dependent Gaussian random distribution in Figure 9.1.

The square area 'wraps around' at the top and bottom and the sides. Sensor nodes at opposite edges will have just a small distance between them. The 'wrapped' network mimics a network of infinite size when the node's transmission range is much smaller than the area dimensions. We use this method to eliminate the effects of nodes at the edges with reduced connectivity, and make the performance characteristics independent of network size. In all of the experiments described we have experimented with networks of different numbers of nodes, and found the results to be independent of network size.

We perform our simulations with varying network densities, measured as the network's connectivity. We define the *connectivity* of the network as the average number of other nodes that each node is able to send to or receive from with reception probability of 50 % or greater. Varying the area size on which nodes are placed randomly has the effect of increasing or decreasing connectivity. Connectivity is displayed on the on of the horizontal axes of the graphs in Figures 9.2, 9.3 and 9.5.

All simulations are performed with large networks of 700 nodes, to reduce the influence of random effects, but extensive testing has shown the results to be representative of smaller networks as well.

Using the described network model the simulation results are independent of parameters such as transmission power, receiver sensitivity and separation between nodes. Similarly, our simulations are independent of timing-specific parameter parameterizations by making the following assumptions:

- The duration of a packet transmission is modeled to be zero, we measure only the occurrence of packet transmissions and receptions.

- Our experiments evaluate an application that is periodical in nature. The duration of these periods is irrelevant, as communication is assumed instantaneous.

- Packet collisions do not occur (and would be impossible with instantaneous communication).

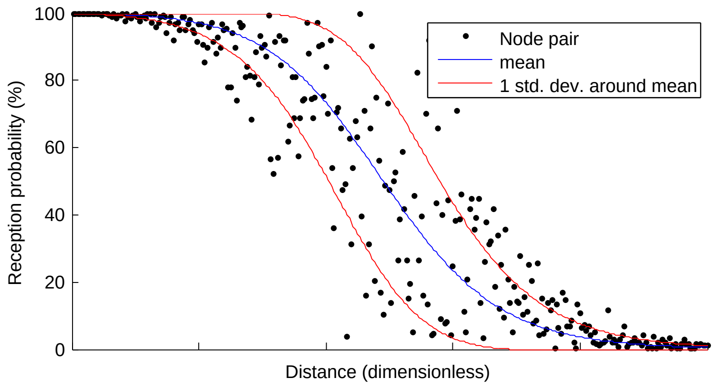


Figure 9.1: Random distribution of packet reception probability vs. node distance generated from communication model. The lines shown are the distribution means, plus and minus one standard deviation.

Though abstract, these assumptions are realistic for the applications evaluated. They accurately mimic real networks with very long periods, short packet transmission times and a randomly chosen moment of transmission within each period, such that no collisions will occur. When using shorter periods, communication intensifies and the probability of packet collisions increases. The goal of avoiding packet collisions coincides with our goal of reducing communication: reduction of communication results in a proportional reduction of packet loss, and increased communication results in an increase in packet loss due to collisions.

Communication model

Our experiments use a communication model based on Ganesan's empirical data [GKW$^+$02] that relates inter-node distance to reception probability. Sampling the mean and standard deviation of the reception rate at various distances yields probability distributions of packet receptions as a function of transmitter-receiver distance.

Figure 9.1 shows 300 samples taken from our communication model. It shows the packet reception rate between pairs of nodes (the black dots) at various distances between sender and receiver. Additionally, the figure contains lines plotting the parameters of our communication model: the distribution mean as a function of distance, and mean plus and minus one standard deviation.

The communication model is dimensionless on the inter-node distance. This means that the distance axis scales for different set-ups of networks, depending on physical parameters such as transmission power, antenna impedance as well as network density. Still, the reception probability as a function of inter-node distance will have the same shape. This generic communication model is independent of physical parameters other than the distance between nodes.

9.3.2 Two-hop gossip

The first test evaluates the impact of ObjectStreams' multi-packet messaging on packet reception and memory use in a broadcast setting. We compare a two-hop gossip application modified from the intruder detection scenario in Listing 5.5 with two alternative implementations that represent a similar effort from programmers. The alternatives are written for TinyOS 2 and only use communication and memory management facilities provided by TinyOS 2.

The goal for the three implementations under test is to deliver a single sensor measurement to all nodes in the 2-hop neighborhood, while minimizing both the wireless communication and memory use to store or buffer data received from the network. The applications use only broadcast communication without retransmissions.

Listing 9.1 shows the SensorScheme implementation we use here. Each round it broadcasts its own sensor value along with values received from direct neighbors in the previous round in a single message, which may span several packets, depending on the number of neighbor values to transmit. When a node receives such a message, it stores the first item – the sending node's sensor value, to transmit it in the next round. The other data items in the message are the sensor values from two-hop neighbors.

Listings 9.2 and 9.3 show code snippets for the alternative implementations. They use different algorithms to implement the same application. The first alternative (Listing 9.2, called *single-packet*) transports a single value from one neighbor per packet. Upon reception of single hop neighbor data, nodes directly forward it to their neighbors. This method requires no buffering, at the expense of broadcasting a large number of small packets. Various implementations exist of this strategy, among others the n–hop *abstract region* [WM04].

The second (Listing 9.3, named *multi-packet*) minimizes communication by using large statically allocated message buffers. This implementation is similar to the Rime Rudolph communication protocol [Dun07] (part of the Contiki OS

```
(define (time-loop t)
  (call-at-time (+ t 16) time-loop)
  ...
  (bcast (msg neigh-msg (cons id (sense-mag))))
  (set! neigh-ls ()))

(define-handler (neigh-msg ls)
  ; add direct neightbor's data to neigh-ls
  (set! neigh-ls (cons (car ls) neigh-ls))
  (process-neighs (cdr ls)))
```

Listing 9.1: Communication in SensorScheme implementation

[DGV04]). As nodes receive messages from their neighbors, they store and buffer data items from their direct neighbors into these large buffers. At the end of each period they send the content of this message buffer as multiple packets. During each period nodes are receiving multi-packet messages from multiple neighbors, and each needs to be stored in a separate message buffer. The number of message buffers available determines the maximum number of messages that can be received simultaneously.

All three implementations send data items containing three 16-bit sensor values (sensor value and two coordinates) and a 16 bit network address. ObjectStreams encodes its payload using 4×20 bits to encode four 16 bit integers $+$ 4 bits for opening and closing brackets, resulting in 84 bits per neighbor value. The single-packet implementation always sends a single neighbor value per packet. Multi-packet uses 4×16 bits $= 64$ bits of payload per neighbor value.

The experiment simulates all three applications multiple times for networks of varying densities and with packet sizes between 28 and 120 bytes of payload. Each simulation run uses a network of 700 nodes for a duration of 20 rounds and averages the number of packets sent and received per node, the memory required to receive these packets and the number of two-hop neighbors from which data is received.

Packets sent and received

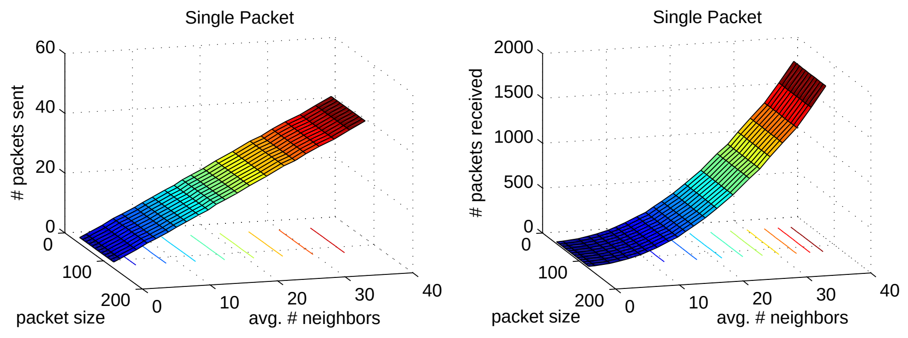
Figures 9.2 and 9.3 show the evaluation results of these three protocol implementations for varying connectivity rates and packet sizes. Figure 9.2 shows the number of packets sent (on the left) and the packets received (on the right) per period for each of the three alternatives. The graphs display how the number of packets sent and received depend on network density. Per round nodes send

```
event PeriodTimer.fired():
  packet_t *pkt = BufferPool.get()
  SensorValToPacket(pkt)
  HopOne.send(pkt)

event HopOne.receive(packet_t *pkt):
  ProcessNeighborData(pkt)
  HopTwo.send(msg)

event HopTwo.receive(packet_t *pkt):
  ProcessNeighborData(pkt)
```

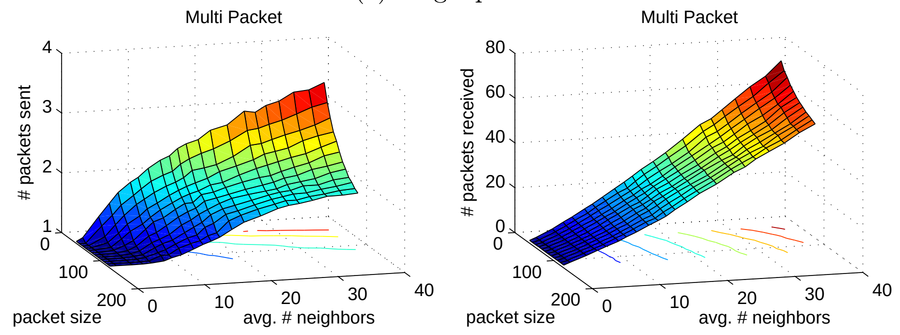
Listing 9.2: *Single-packet* alternative implementation, directly forwarding received neighbor values

```
event PeriodTimer.fired():
  MultiPacket.send(SendBuffer, SendCount)
  SendBuffer[0] ={myAddress, sensorVal}
  SendCount = 1

event SinglePacket.receive(packet_t *pkt, int len) {
  if isStartPacket(pkt):
    s = newSeqRecord(pkt)
    s.buf = BufferPool.get()
    s.rcvPtr = 0
  else:
    s = findSeqRecord(pkt)
  s.buf[s.rcvPtr:s.rcvPtr+len] = getPayload(pkt)
  s.rcvPtr += len
  if isEndPacket(pkt):
    signal MultiPacket.receive(s.buf, s.rcvPtr)

event MultiPacket.receive(buffer_t *buf, int len):
  // senderÕs data in first item
  SendBuffer[SendCount]  = buf[0]
  SendCount += 1
  for i in 0 .. len :
    ProcessNeighborData(buf[i])
```

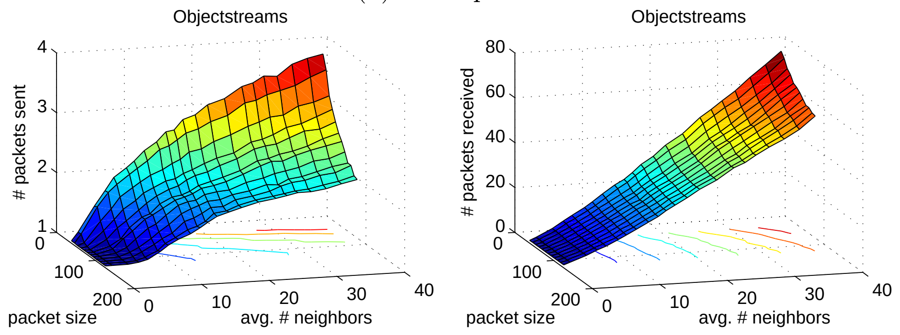
Listing 9.3: Alternative implementation, using intermediate buffering



(a) Single-packet

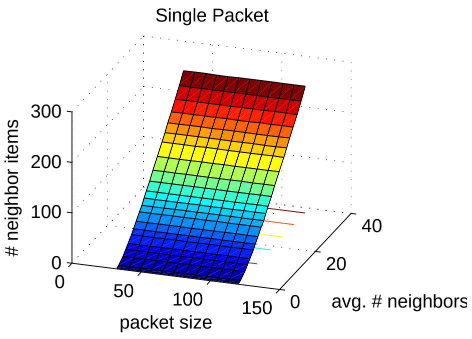


(b) Multi-packet

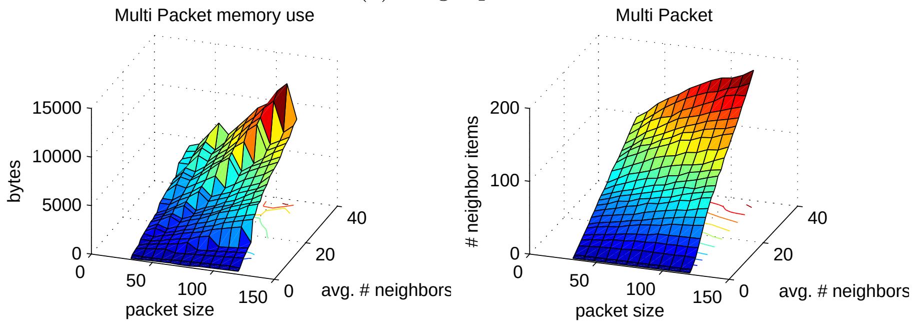


(c) ObjectStreams

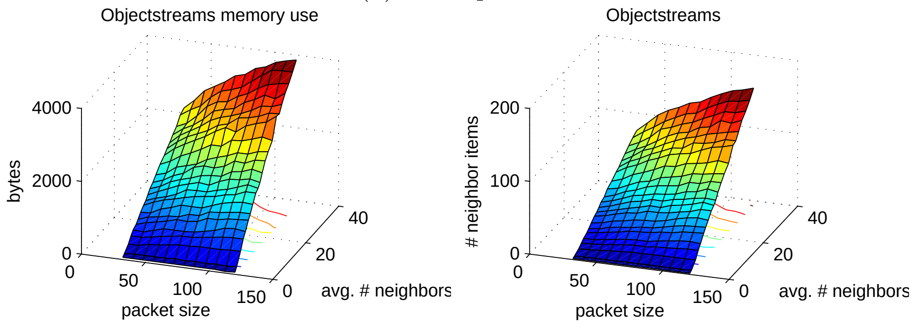
Figure 9.2: Evaluation results of two-hop gossip application: Number of packets sent (left) and received(right).



(a) Single-packet



(b) Multi-packet



(c) ObjectStreams

Figure 9.3: Evaluation results of two-hop gossip application: Memory use (left) and unique data items received (right).

one data item per direct neighbor and receive all data sent by their neighbors.

The single-packet algorithm represents the worst case scenario with linear resp. quadratic increase of sent resp. received packets for increasing density. Note that the single-packet data is independent of packet size. The content per packet is fixed, which quickly raises the communication needs for more dense networks.

The multi-packet and ObjectStreams versions can significantly reduce the number of packets sent and received by sending multiple data items per packet. The number of packets sent and received depends on packet size: larger packets require fewer packets to be sent. Communication requirements for multi-packet and ObjectStreams are similar. Individual neighbor items are packed as 64 bits (4 16 bit values) in multi-packet and 80 bits for ObjectStreams, a 25 % increase. ObjectStreams sends on average about 12 % more packets. Similarly, ObjectStreams receives about 5 % more packets.

Memory requirements

A second performance metric is the amount of memory needed by each application. Figure 9.3 shows the memory required by each version on the left. Figure 9.3 (a) does not show the memory requirements of Single-packet, as it has memory requirements of only a single packet.

The multi-packet version requires multiple message buffers to receive and store received neighbor data items, one per connection to a neighbor. We calculate the memory required for Multi-packet by multiplying the number of message buffers and the buffer size required to store all data received in a round. Each message buffer is the size of one or more network packets.

Figure 9.3 (b) and (c) shows (in the left graphs) the memory required for both multi-packet and ObjectStreams. We measure the amount of memory required to store all data received by a node in a single round. The amount of data received differs per node, depending on the number of neighbors. Our memory measurements record the memory required to completely store the data received by the lowest 95 % of nodes; with the amount of memory reported, 5 % of nodes will not have enough memory to store all the packets they are able to receive.

The graphs show that ObjectStreams requires less than half the memory required by multi-packet. Even though ObjectStreams stores all received data as linked lists in memory, still it requires less memory than the pre-allocated memory buffers required for the Multi-packet version.

Unique data items

We measure the application performance as the number of unique neighbor items that each node receives per period. Nodes may receive items from two-hop neighbors from multiple single hop neighbors, but the intruder detection application requires data items from a two-hop neighbor only once. Figure 9.3 shows (on the right) the number of unique data items received per period. Their differences are mostly due to the influence of packet loss. In the single-packet implementation, when a packet from a neighbor does not arrive, only a single neighbor value is lost. Using the SensorScheme and multi-packet implementations, when a packet is not received, the entire multi-packet message it is part of is discarded, and the data therein is lost. When a message is lost, its first-hop data is also not available for retransmission, which reduces communication even further for denser networks.

Concusion

From these results we can conclude that the three implementations behave quite differently: The single-packet implementation has the lowest memory footprint and achieves the best application performance (in terms of the number of distinct data items received). The multi-packet implementation is the most efficient as far as communication is considered, at the cost of considerable memory requirements. In between these two, ObjectStreams operates more energy-efficient than the Single-packet solution, while requiring less memory to operate than Multi-packet.

For this particular application we conclude that using SensorScheme and the ObjectStreams communication mechanism strikes a balance between energy efficiency and memory required.

9.3.3 Tree–routed data collection

The second evaluation of the ObjectStreams communication method uses a tree routing protocol to transport data from all nodes in the network to a single root. The environmental monitoring application described in Section 3.2 assumes this transport method, and the various implementations such as SwissQM [MRD$^+$07] and TinyDB [MFHW03] use it as well.

The prime appeal of this method of environmental monitoring is the possibility of sensor data aggregation, as it reduces communication to a constant amount of data to be transmitted from every intermediate node of the tree. Aggregation does produce only a summary of the sensed data (such as the average, minimum or maximum) instead of the full data set. When this is not desired, intermediate nodes need to forward all individual data values received from nodes higher up in the tree, increasing the traffic load at the bottom nodes. In current implementations, such as SwissQM [MRD$^+$07] and TinyDB [MFHW03], every individual sensor data item is transmitted in a separate packet, which requires the number of packets to be received by the gateway node to be equal to the network size. This sheer volume of communication overloads the wireless medium with only modestly sized networks, and quickly drains the batteries of nodes lower in the tree.

When all data from the entire network is requested, the data has a high degree of redundancy, both in time and in space, something that is made use of in implementations such as TinyDB and SwissQM to cope with the unreliability and low bandwidth of the wireless network.

Each sensor data packet is sent to the node's parent node in a best–effort fashion, without retransmissions in the case of packet loss. The result is that only a fraction of the sensor data is actually delivered to the routing tree. The probability of delivery is not equal for all nodes, but depends on the position in the routing tree. For nodes high up in the routing tree, delivery of its data takes many individual transmissions, each of which may fail.

To graphically indicate the magnitude of both of these issues, figure 9.4 shows a randomly generated routing tree configuration for a sensor network of 800 nodes. The circles show the locations of the nodes, and the lines between them indicate a parent link for each node. The tree root is a node in the center of the figure, colored white. The color of the other circles indicate the end-to-end delivery probability of each node's sensor data, ranging from as low as 16 % (in blue) in the outer corners of the network to 100 % for nodes near the root (in red). The line thickness of the links connecting nodes to their parents are proportional to the amount of data traveling across it. The connections near the base of the network carry a heavy load, since these have to transport all data from higher up in the network.

The SensorScheme implementation of the environmental monitoring application addresses this application somewhat differently. Our second query in Section 5.7 uses an aggregation function `fold` that folds all received data into a list, packing the received data in a single large message that may span multiple packets. Transmission of a single multi-packet message to a node's parent may reduce communication significantly compared to individual packets for every data value, as other implementations do. The sensor values produced by each node are small, and a concatenated message can contain many sensor values per underlying packet, reducing the communication load to a fraction of that used

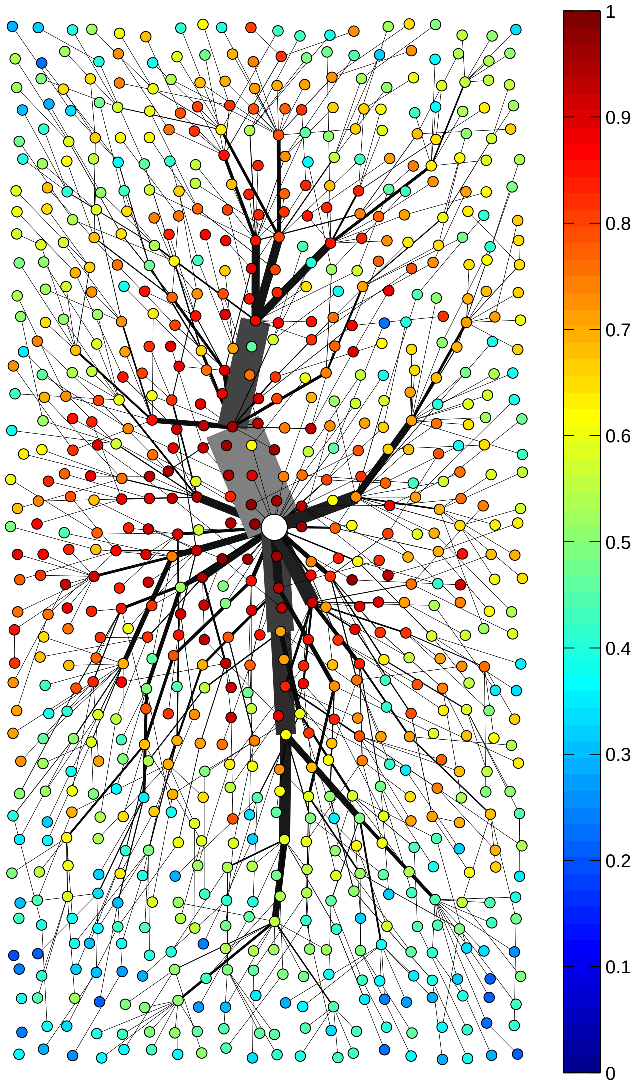


Figure 9.4: Example of a routing tree configuration for a sensor network of 800 nodes.

in the current solutions.

The SensorScheme solution further uses the reliable communication protocol described in Section 7.5.2 to communicate between nodes and their parents. This reduces the problem of data loss due to long and unreliable links to the root, and increases data reception to near 100 %.

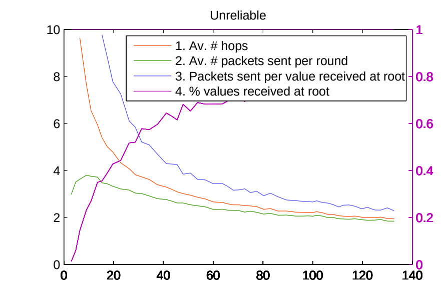
Our next experiment evaluates the reliable tree collection ObjectStreams implementation. We evaluate its performance by comparing it to two alternative implementations. Both alternative implementations use a single packet for every sampled data set, similar to TinyDB [MFHW03] and SwissQM [MAK07]. The first alternative – called *unreliable* – transports the sensor data in a best-effort manner, without the use of acknowledgements and retransmissions. The other implementation, called *retransmit* uses acknowledgements and retransmissions similar to the SensorScheme implementation: Parent nodes will send an acknowledgement upon reception of packets containing sensor data items. When the sender does not receive the acknowledgement, it retransmits the packet up to 5 times, and reselects a parent if the fifth retransmission was not successful either, after which transmission restarts. All three protocols use the TinyOS 2 Collection protocol [Netb] to create and maintain the routing tree. The communication involved in tree construction and maintenance is not included in the measurements reported here.

The goal of the tree collection protocol is to transport sampled sensor data to the root of the routing tree. We compare the three implementations on two properties:

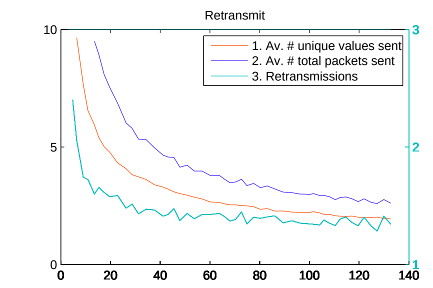
1. The energy-efficiency of delivering individual measurements to the collection root. Again, we measure the number of packets sent and received to determine energy-efficiency.

2. The root node is the bottle-neck for a data collection application, and determines maximum data collection frequency and scaling parameters. We therefore compare the communication volume to the collection root.

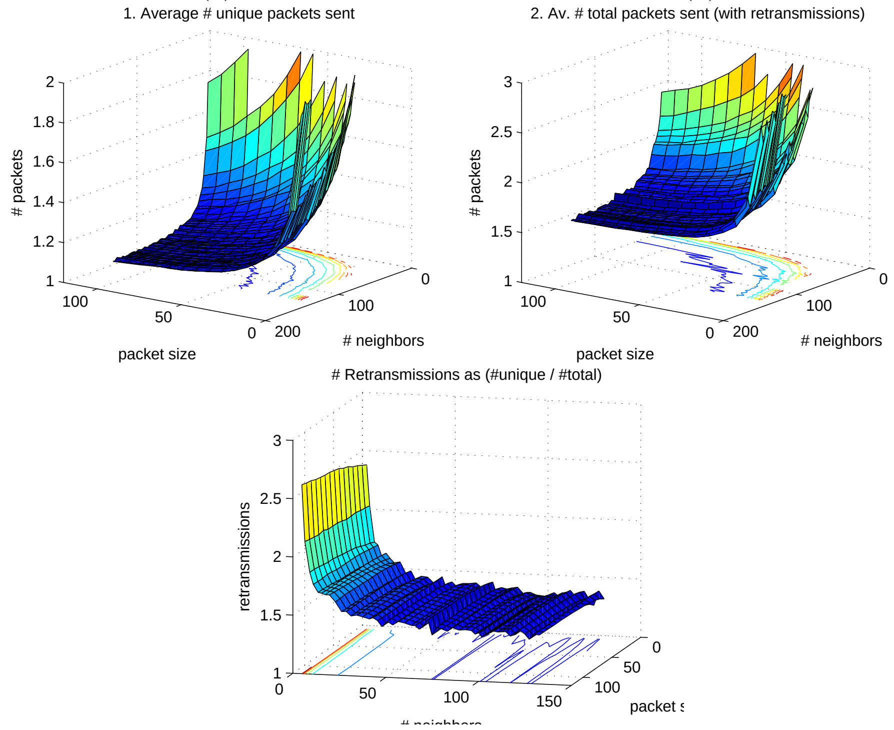
Before discussing the comparative results we first discuss some differences between the used implementations. Most importantly, both ObjectStreams and *retransmit* use reliable communication, whereas *unreliable* uses only a best-effort data transport. In case of a packet reception failure from a leaf node to its parent, *unreliable* will not be able to deliver the sensor data to the network root, and hence, a certain percentage of measurements is lost due to unreliable communication. The ObjectStreams and *retransmit* implementations achieve a near 100 % delivery ratio of measurements to the root.



(a)



(b)



(c)

Figure 9.5: Performance results of the tree routing protocol evaluation.

We have again run simulations with 700 nodes, with different network densities as show in the horizontal axis in the graphs of Figure 9.5, and in the case of the ObjectStreams with varying packet sizes between 28 and 120 bytes of payload. During this experiment, the simulation first runs for some time while the network forms a routing tree. After the routing tree stabilizes, we again obtain data averaged over 20 rounds.

Figure 9.5 (a) shows the results of running *unreliable* on networks of varying density. The different network densities result in different routing trees. Most important to these results, the tree depth, measured as the average number of hops to the root, is inversely proportional to the network connectivity, as shown in (a).1. The probability of values sent in a best-effort fashion being delivered to the root depends directly on the depth of the root and the reception probability of the individual links between nodes and their parent. Graph (a).4. shows the percentage of values sent from each node that reaches the collection root (using the right-side axis). The number of packets nodes send per round is shown in (a).2. From these values we obtain a measure of efficiency as the number of messages (averaged over all nodes) sent per value delivered at the root, shown in graph (a).3.

The *retransmit* implementation operates similarly, with the exception that communication between a node and its parent is acknowledged and retransmitted until successful according to the method described above. In the absence of bandwidth and buffer limitations in our simulations, communication between nodes and the root is effectively 100 % reliable.

Figure 9.5 (b)1. shows the average number of values (re)transmitted per period, which is equal to the average humber of hops, shown in Figure 9.5 (a)1. Due to packet loss and retransmissions, nodes send a larger number of packets per period, shown in (b)2. The number of retransmissions, obtained by $(b)2/(b)1$ is show in (b)3 (using the right-side axis). Our efficiency metric of the average number of packets sent per value received at the root is represented by (b)2. Interestingly, *unreliable* and *retransmit* perform practically identical.

The two alternative approaches are not dependent on packet size since both transmit only a single measurement per packet, and 2D graphs sufficiently convey their performance characteristics. The ObjectStreams characteristics are dependent on packet size again and require 3D presentation again. Figure 9.5 (c) presents ObjectStreams performance. Analogous to Figure 9.5 (b) Figure 9.5 (c)1, 2 and 3 show resp. the unique number of packets sent by ObjectStreams, the number of packets sent including retransmissions, and the fraction of these two, indicating the frequency of retransmissions. As is to be expected, the results are similar in to the *retransmit* alternative, but requiring fewer packets

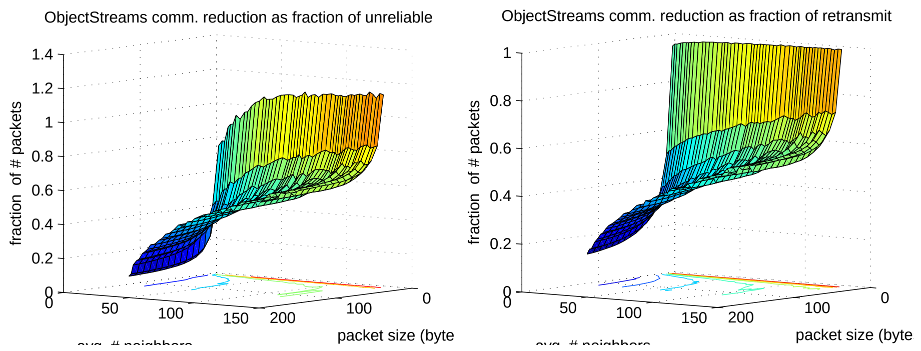


Figure 9.6: Performance results of the tree routing protocol evaluation.

to transmit. Except for sparse networks and small packet sizes, ObjectStreams requires just above 1 packet per value received at the root. Including retransmissions as a result of unreliable wireless links, this amounts to about 1.5 packets per received value.

Figure 9.6 offsets ObjectStreams' performance with both alternatives by dividing ObjectStreams' results by either alternative. The graphs show that for most networks, ObjectStreams is able to reduce communication by 50 % or more, depending on the network density.

SensorScheme requires considerably less communication because it is able to transport multiple sensor measurements in each packet. For sparser networks, where nodes are at greater distances from each other, the tree construction algorithm creates deep trees, consisting of connections to parent node that have a high probability of failing. Denser networks need less communication, as a result of lower packet loss rates and reduced tree depth. Figure 9.5 (b) shows improvement SensorScheme makes over the other implementations by dividing the the *unreliable* and *retransmit* graph data of (a) by the SensorScheme data.

9.3.4 Conclusion

We have compared the cost of communication when using ObjectStreams to alternative implementation strategies for two different communication protocols: the two-hop gossip protocol used in the intruder detection application and the collection protocol used with environmental monitoring. We have selected protocols that are described in the state of the art, and that would require a comparable implementation effort, using general-purpose communication protocols such as abstract regions [WM04], the Contiki Rime stack [Dun07] or the TinyOS Collection protocol [Netb]. We have taken a look at how each of the alternatives compare in terms of communication cost measured as the number of messages transmitted and received and the memory needed to run the application as well as application performance measured as the number of data items delivered to the application.

In the case of the two-hop gossip protocol ObjectStreams holds the middle between the two alternatives. Objectstreams is somewhat more expensive in terms of communication than the more efficient alternative, *multi-packet*. On the other hand, ObjectStreams requires considerably less memory. They deliver almost equal application performance. *Single-packet* represents the opposite side of the spectrum with extreme memory-efficiency and superior application performance exchanged for significant communication cost. From these observations one might argue that ObjectStreams performs satisfactorily for real applications where trade-offs need to be made on energy-efficiency as well as memory use.

For this particular application it might well be possible to construct a protocol that is even more efficient, both regarding communication and memory by building application-specific data structures and communication protocols. This will be, however, at the expense of considerable more application construction time and loosing SensorScheme's reprogrammability and safety.

The second experiment, collecting data from the entire network shows the potential reduction in communication by 50 % or more with the use of multi-packet encoding of variably sized data. The nature of the application, where the structure and size of data from each node is determined at run time makes the use of multi-packet messages non-trivial for implementations such as TinyDB and SwissQM, and even more so for QueryVM. The use of ObjectStreams to encode multi-packet messages clearly shows the benefits of SensorScheme: it supports a wider variety of queries, as we discussed earlier, and in the same time can reduce the communication requirements.

Chapter 10

Conclusions

This dissertation described the SensorScheme software platform and its design principles, motivated by four real world application scenarios. This work has reviewed the state of the art of tools and techniques in use on wireless sensor network platforms and discussed their applicability for the example platform scenarios. The SensorScheme platform extends the state of the art, introducing new techniques to build short and efficient wireless sensor network programs.

To recapitulate on the contributions stated in the introduction (Chapter 1) we can conclude the following:

1. We have shown that the SensorScheme platform enables wireless loading and programming of wireless sensor networks. As the state of the art shows, providing wireless programming on small WSN platforms comes at the price of reduced execution efficiency and increased memory use. SensorScheme is no exception in this respect, but is able to use Scheme's homo-iconic program representation to transfer and execute programs with minimal additional complexity.

   The SensorScheme's design ensures a small and efficient implementation, and shows that a combination of features such as wireless program loading, platform independence, a safe execution environment, garbage collection, and concurrency and blocking I/O are achievable targets even for low power WSN platforms. Furthermore, a combination of SensorScheme's programming techniques, program representation and network encoding ensure that SensorScheme applications are small and efficient to transport across the network, in typically just a few hundreds of bytes.

As the comparison for environmental monitoring shows, using SensorScheme one is able to reduce code size of applications while adding functionality and flexibility. Still, energy use of the resulting program is roughly the same.

Furthermore, we have shown that SensorScheme causes only marginal additional energy use and no significant delays due to program interpretation and garbage collection.

2. SensorScheme's other contribution is the use of a dialect of Scheme to write sensor network application. Using the application scenario's this thesis explores the ways in which sensor network applications may be built with the availability of a programming language that includes first class functions and closures, and continuations. We have shown that sensor network applications are particularly suitable to a functional programming approach. Furthermore, closures in SensorScheme are a powerful program structuring technique, obviating the need of object-oriented language features. The availability of continuations enables the use of co-routines, which allows programs to be structured similar to multi-threaded applications, including the use of blocking I/O calls. All of these program structuring techniques make it possible to drastically reduce the size of WSN applications.

3. SensorScheme includes the ObjectStreams communication mechanism, unique in the sensor network field of research, which again aid in the construction of a wide variety of applications while keeping them short and easy to build.

   ObjectStreams transfers messages that may be larger than single packets, which makes applications independent of the packet size of the target platform, and simplifies communication of large or variably-sized payloads. Furthermore, ObjectStreams transports data structures rather than simply arrays of bytes, which eliminates the need for application code to pack and unpack message content. ObjectStreams uses a compressing serialization method to efficiently transport data of arbitrary size and structure.

   ObjectStreams transports application data as well as program code in a uniform way, as a result of SensorScheme's unique property that program code is just a special case of application data, and can thus be treated equally.

4. SensorScheme uses partial evaluation to specialize a general program describing the behavior of the entire network into node-specific variants that are significantly shorter. We use this specialization method as a new way to macro-program heterogeneous sensor networks.

   The main benefit of this program deployment procedure for heterogeneous WSNs is that the individual specialized programs can be much smaller than the generic network-wide program. This leads to decreased communication use during program transfer, and may significantly reduce memory requirements. Furthermore, our method merges program logic and configuration data, so installing a new program and configuring the network takes place using the same injection mechanisms built-in in the SensorScheme platform, (thereby eliminating the need for separate mechanisms of reconfiguring the devices). This in turn reduces the total program size even further, and simplifies development and operation of WSN applications.

Appendix A

Formal definition

```
(ssmodule eval
  (require "thesis-base.ss")

  (define (my-eval expr env)
    (cond
      [(or (number? expr)
           (boolean? expr)
           (null? expr))   expr]
      [(symbol? expr)      (lookup expr env)]

      [(eq? (car expr) 'quote)  (cadr expr)]
      [(eq? (car expr) 'if)     (if (not (eq? (my-eval (cadr expr) env) #f))
                                    (my-eval (caddr expr) env)
                                    (if (not (null? (cdddr expr))) (my-eval (cadddr expr) env) #f))]
      [(eq? (car expr) 'lambda) (cons 'proc (cons env (cdr expr)))]
      [(eq? (car expr) 'set!)   (update! (cadr expr) (my-eval (caddr expr) env) env)]
      [(eq? (car expr) 'define) (extend! (cadr expr) (my-eval (caddr expr) env))]
      [else
       (my-apply (my-eval (car expr) env) (map (lambda (e) (my-eval e env)) (cdr expr)))]))

  (define (my-apply proc args)
    (cond [(primitive? proc) (apply-prim proc args)]
          [(eq? (car proc) 'proc)
           (eval-body (cdddr proc)      ; proc.  body
                      (bind (caddr proc) ; parameters
                            args
                            (cadr proc)))]))  ; captured environment


  (define (eval-body exprs env)
    (if (null? (cdr exprs)) ; last expression, return its result
        (my-eval (car exprs) env)
        ((lambda ()
           (my-eval (car exprs) env)
           (eval-body (cdr exprs) env)))))

  (define (bind vars vals env)
    (cond [(and (null? vars) (null? vals)) env]
          [(and (pair? vars) (pair? vals) (symbol? (car vars)))
           (cons (cons (car vars) (car vals))
                 (bind (cdr vars) (cdr vals) env))]
          [(symbol? vars) (cons (cons vars vals) env)]
          [else (error 'parameter-mismatch)]))

  (define (lookup var env)
    (let ([l (assoc var env)])
      (if l (cdr l)
          (let ([g (assoc var global-env)])
            (if g (cdr g)
                (error 'variable-not-found))))))

  (define (update! var val env)
    (let ([l (assoc var env)])
      (if l (set-cdr! l val)
          (let ([g (assoc var global-env)])
            (if g (set-cdr! g val)
                (error 'variable-not-found))))))

  (define (extend! var val)
    (let ([v (assoc var global-env)])
      (if (pair? v)
          (set-cdr! v val)
          (set! global-env (cons (cons var val) global-env)))))

  ; use the eval definition by calling it
  (define global-env '([cons . (prim cons)]
                       [car . (prim car)]
                       [cdr . (prim cdr)]))

  )
```

Appendix B

Solutions

B.1 Intruder detection

```
(ssmodule intruder-twohop
  (require "thesis-base.ss")

  (define-const threshold 15)

  ; return node value with maximum reading
  (define (max-node l r)
    (if (> (second l) (second r)) l r))

  (define (process-neighs ls)
    (unless (or (null? ls) (member (caar ls) id-ls))
      (set! id-ls (cons (caar ls) id-ls))
      (max-node-fold (car ls) #f)
      (sum-v-fold (second (car ls)) #f)
      (sum-x-fold (third (car ls)) #f)
      (sum-y-fold (fourth (car ls)) #f)
      (process-neighs (cdr ls))))

  (define-handler (neigh-msg ls)
    ; add direct neightbor's data to neigh-ls
    ; only if node seds its own data
    (when (= src (caar ls))
      (set! neigh-ls (cons (car ls) neigh-ls)))
    ; calculate partial max-node and centroid
    (process-neighs (cdr ls)))

  (define neigh-ls ())
  (define max-node-fold (closure-fold max-node ()))
  (define sum-v-fold (closure-fold + 0))
  (define sum-x-fold (closure-fold + 0))
  (define sum-y-fold (closure-fold + 0))
  (define id-ls ())
  (define (time-loop t)
    (call-at-time (+ t 16) time-loop)
    (let* ([v (sense-mag)]
           [me (list id v (* v (x-coord)) (* v (y-coord)))]
           [sum-v (sum-v-fold 0 #t)]
           [sum-x (sum-x-fold 0 #t)]
           [sum-y (sum-y-fold 0 #t)]
           [msg-ls (if (> v threshold) (cons me neigh-ls) neigh-ls)])
      (set! neigh-ls ()) ; reset list of received neighbors
      (max-node-fold me #f) ; add own data
      (when (eq? me (max-node-fold () #t))
        (send-root (msg neigh-result (list sum-v (/ sum-x sum-v) (/ sum-y sum-v)))))
      (bcast (msg neigh-msg msg-ls))))

  (time-loop (now))

  )
```

Listing B.1: Two hop intruder detection program.

B.2 Environmental monitoring

```
(ssmodule monitoring
  (require "thesis-base.ss")

  ; definitions of query
  (define period (* 30 16))
  (define duration (* 30 16 60))

  (define (init)
    (if (= (+ (/ (x-coord) 10) (* (/ (y-coord) 10) 4)) 6)
        (list (list id (sense-temp)))
        ()))

  (define (proc l r)
    (append l r))

  ; auxillary functions
  (define (current-epoch)
    (/ (synced-now) period))

  (define recv-fold (closure-fold proc (init)))

  ; handler called when receiveing message from children
  (define-handler (parent-msg val)
    (recv-fold val #f))

  (define (time-loop t)
    (unless (> (+ start-time duration) (synced-now))
      (call-at-time/synced (+ t period) time-loop))
    (let ([init-val (init)])
      (call-at-time/synced (+ t (* (- 8 (hops-to-root)) 16))
                    (lambda (t)
                      (let ([agg-val (recv-fold init-val #t)])
                        (unless (null? agg-val))
                      (send-parent (msg parent-msg agg-val)))))))

  (define start-time (synced-now))
  (time-loop (current-epoch))

  )
```

Listing B.2: Environmental monitoring application.

```
(ssmodule ssquery
  (require "thesis-base.ss" )

  (define query-ls ())

  (define-handler (parent-msg qid val)
    ; call handle-parent-msg of query 'qid'
    ((cdr (assoc qid query-ls)) val))

  (define-handler (query-msg qid period duration proc init)
    (let ([recv-fold (closure-fold (eval proc) ((eval init)))]
          [start-time (now)])
      (define (handle-parent-msg val)
        (recv-fold val #f))

      (define (current-epoch)
        (/ (now) period))

      (define (time-loop t)
        (if (> (+ start-time duration) (now))
            (set! query-ls (filter (lambda (el) (not (eq? el qid))) query-ls))
            (call-at-time/synced (+ t period) time-loop))
        (let ([init-val ((eval init))])
          (call-at-time/synced (+ t (* (- 8 (hops-to-root)) 16))
                               (lambda (t)
                                 (let ([agg-val (recv-fold init-val #t)])
                                   (unless (null? agg-val))
                                   (send-parent (msg parent-msg agg-val)))))))

      (set! query-ls (cons (cons qid handle-parent-msg) query-ls))
      (time-loop (current-epoch))))

  )
```

Listing B.3: SSQuery

B.3 Logistics

```
(ssmodule logistics
  (require "thesis-base.ss" )
  (define alerter print)
  (define (blink-leds col)
    (case col
      [(red) 1]
      [(green) 2]
      [(blue) 4]))
  (define (send/ack n m dst msg) msg)

  (define state 0)
  (define (log-state-change new-state)
    (set! state new-state)
    (log (now) new-state ()))

  (define (msg-filter type key val? ls)
    (filter (lambda (el)
              (and (eq? (second el) type)
                   (let ([v (assoc key (third el))])
                     (if v (val? (cdr v)))))) ls))

  (define msg-ls ())
  (define-handler (logistics-msg type kvlist)
    (set! msg-ls (cons (list src type kvlist) msg-ls)))

  (define properties ())
  (define (set-property! key val)
    (set! properties (cons (cons key val) properties)))

  (define (state-loop alarm? alarmer transit?)
    (call/cc
     (lambda (k)
       (define (time-loop t)
         (if (transit? msg-ls) (k #t)
             (begin
               (call-at-time (+ t 80) time-loop)
               (when (alarm? msg-ls) (alerter msg-ls))
               (set! msg-ls ())
               (bcast (msg logistics-msg 'goods
                           properties)))))
       (time-loop (now))
       (k #f))))

  ; coffee checker definitions
  (define (coffee-checker t)
    (call-at-time (+ now 160))
    (bcast (msg content-msg 'bananas)))

  (define-handler (content-msg content)
    (when (eq? content 'coffee)
      (blink-leds 'red)
      (bcast (msg alarm-msg 'dangerous-content 'coffee))))

  ;;;;; main program
  ; process 1
  (coffee-checker (now))

  ;process 2
  (call-at-time (now) (lambda (t)
    (set-property! 'content 'bananas)
    ; 1.  at farm
    (log-state-change 'farm)
    (set-property! 'dest 'Rio-harbor)
    (state-loop
     ; alarm-condition:  detected peers with different destination
     (lambda (ls)
       (not (eq? ()
                 (msg-filter 'goods 'dest
                             (lambda (v)
                               (not (eq? v 'Rio-harbor))) ls))))
     ;alarm-procedure:  blink red leds
     (lambda (ls) (blink-leds 'red))
     ; transition-condition:  detect truck with destiation Rio-harbor
```

```
 (lambda (ls) ; list of trucks with destination is not empty
   (not (eq? ()
             (msg-filter 'transport 'dest
                         (lambda (v)
                           (eq? v 'Rio-harbor)) ls)))))

; 2.  in truck
(log-state-change 'truck)
(state-loop
 ; alarm-condition:  truck not found anymore
 (lambda (ls)
   (eq? ()
        (msg-filter 'transport 'dest
                    (lambda (v)
                      (not (eq? v 'Rio-harbor))) ls)))
 ; alarm-procedure:  blink red leds
 (lambda (ls) (blink-leds 'red))
 ; transition-condition:  detect harbor and absence of truck
 (lambda (ls)
   (and (not (eq? ()
                  (msg-filter 'infrastructure 'location
                              (lambda (v)
                                (eq? v 'Rio-harbor)) ls)))
        (eq? ()
             (msg-filter 'transport 'dest
                         (lambda (v)
                           (not (eq? v 'Rio-harbor))) ls)))))

; 3.  on harbor dock
(log-state-change 'harbor)
(set-property! 'dest 'Roterdam-distcenter)
(state-loop
 ; alarm-condition:  not on right dock position?
 (lambda (ls)
   (eq? ()
        (msg-filter 'infrastructure 'dock-nr
                    (lambda (v)
                      (eq? v 364))
                    (msg-filter 'infrastructure 'location
                                (lambda (v)
                                  (eq? v 'Rio-harbor)) ls))))
 ; alarm-procedure:  send message to dock access point
 (lambda (ls) (send/ack 3 5
                        ; send to address of harbor access point
                        (caar (msg-filter 'infrastructure 'location
                                          (lambda (v)
                                            (eq? v 'Rio-harbor)) ls))
                        (msg alarm-msg properties)))
 ; transition-condition:  detect container with right shipping-id
 (lambda (ls)
   (not (eq? ()
             (msg-filter 'transport 'shipping-id
                         (lambda (v)
                           (eq? v '3476353)) ls)))))

; 4.  inside container
(log-state-change 'container)
(state-loop
 ; alarm-condition:  not with peers to go to distcenter
 (lambda (ls)
   (eq? ()
        (msg-filter 'infrastructure 'location
                    (lambda (v)
                      (eq? v 'Roterdam-distcenter)) ls)))
 ;alarm-procedure:  blink red, and send message to container
 (lambda (ls) (blink-leds 'red)
   (send/ack 3 5
             (caar (msg-filter 'transport 'shipping-id
                               (lambda (v)
                                 (eq? v '3476353)) ls)) ; address of container
             (msg alarm-msg properties)))
 ;transition-condition:  detect container with right shipping-id
 (lambda (ls)
   (not (eq? ()
             (msg-filter 'infrastructure 'location
                         (lambda (v)
                           (eq? v 'Roterdam-distcenter)) ls))))))
```

```
  )
; 5.  at distribution center
```

Listing B.4: Itinerary program of the logistics scenario.

B.4 Smart office

```
(ssmodule smart-office
  (require "thesis-base.ss" "office-constants.ss")

  ; auxiliary functions
  (define-const room-nr car)

  (define-const room-coords cdr)

  (define-const node-coords caddr)

  (define (within? node room)
    (= (car node) (car room)))

  (define (room-of nd)
    (room-nr (find (lambda (el)
                     (within? (node-coords (assoc nd nodes-tbl))
                              (room-coords el))) rooms-tbl)))

  (define (nodes-in-room room)
    (map car (filter (lambda (el) (= room (room-of (car el)))) nodes-tbl)))

  (define-const (room-temp room) 25)

  (define (adjust-temp val)
    (bcast (msg adjust-temp val)))

  (define (average ls)
    (/ (foldr + 0 ls) (length ls)))

  (define (node-type node)
    (cadr (assoc node nodes-tbl)))

  ; node type checking functions
  (define (node-is-switch? node)
    (nth 0 (cdr (assoc (node-type node) types-tbl))))

  (define (node-is-temp? node)
    (nth 1 (cdr (assoc (node-type node) types-tbl))))

  (define (node-is-light? node)
    (nth 4 (cdr (assoc (node-type node) types-tbl))))

  (define (node-is-radiator? node)
    (nth 5 (cdr (assoc (node-type node) types-tbl))))

  ; connect a switch to all lights in the room
  (define-event (light-switch on?)
    (when (node-is-switch? id)
      (let ([room (room-of id)])
        (for-each (lambda (el)
                    (send-local el (msg switch-light-hdl on?)))
                  (filter node-is-light? (nodes-in-room room))))))

  (define-handler (switch-light-hdl)
    (when (node-is-light? id)
      (toggle-light!)))

  (define collect-ls ())
  ; use average temperature of room for heating to control radiator
  (define-handler (collect-temp-hdl room-num val)
    (when (node-is-temp? id)
      ; collect protocol
      (if (= room-num (room-of id))
          (set! collect-ls (cons val collect-ls)))))

  (define (sense-temp-loop t)
    (when (node-is-temp? id)
      (call-at-time (+ t (* 16 60)) sense-temp-loop)
      (let ([room (room-of id)])
        (when (= id (max collect-ls))
          (for-each
           (lambda (el)
             (send-local el (msg report-temp-hdl (average (map cdr collect-ls)))))
           (filter node-is-radiator? (nodes-in-room room))))
        (set! collect-ls ())
        (bcast (msg collect-temp-hdl room (sense-temp))))))

  (when (node-is-temp? id)
    (sense-temp-loop (now)))

  ; radiator nodes
  (define-handler (report-temp-hdl val)
    (when (node-is-radiator? id)
      (adjust-temp (- (room-temp (room-of id)) val))))
  )
```

Listing B.5: Generic program of the smart office scenario.

Appendix C

Source and auxiliary files

The following module files are part of the SensorScheme library, and contain the procedure and macro definitions used elsewhere.

C.1 Base library

```
(ssmodule thesis-base

  (provide (all-from "std.ss"))
  ; include all the primitives
  (include id car cdr set-car! set-cdr! cons + - * /
           > >= < <= eq?
           null? pair? symbol? number? boolean? not
           random now call-at-time sense-temp blink toggle-light!
           append list
           print call/cc eval apply)

  ; collection protocol library
  (provide (all-from "collection.ss"))

  ; program injection library
  (require "inject.ss")
  (include recv-handoff)

  ; definitions for example programs
  (provide group closure-fold)

  (define (group fn l r)
    (foldl (lambda (a b)
             (if (assoc (first a) b)
                 b
                 (cons a b)))
           (map (lambda (l-area)
                  (let ([r-area (assoc (first l-area) r)])
                    (if r-area
                        (fn l-area r-area)
                        l-area))) l) r))

  (define (closure-fold kons knil)
    (lambda (el return?)
      (print 'closure-fold knil el return?)
      (if return?
          (begin0 knil (set! knil el))
          (set! knil (kons (car el) knil)))))

  ; networking definitions
  (provide handle send-local bcast msg)

  (define (handle src msg)
    (apply (eval (car msg)) (cons src (cdr msg))))

  (define (bcast mess)
    (send-local -1 mess))

  (define-primitive send-local (sender AMSender))
  (define-primitive recv-local (receiver AMReceiver))

  (define-macro (msg tag . body)
    `(list ',tag ,@body))

  (include send-local recv-local)
  )
```

Listing C.1: base library, defines node configuration and the set of primitives included.

C.2 Standard library

```
(ssmodule std
  (provide (all-from "macros.ss"))
  (provide (all-from "primitives.ss"))

  (provide caar cadr cddr caddr cdddr cadddr
           first second third fourth fifth nth length
           assoc member find foldl foldr for-each map filter
           max =)

  (define (caar l)
    (car (car l)))

  (define (cadr l)
    (car (cdr l)))

  (define (cddr l)
    (cdr (cdr l)))

  (define (caddr l)
    (car (cdr (cdr l))))

  (define (cdddr l)
    (cdr (cdr (cdr l))))

  (define (cadddr l)
    (car (cdr (cdr (cdr l)))))

  (define-const first car)
  (define-const second cadr)
  (define-const third caddr)
  (define-const fourth cadddr)
  (define-const fifth caddddr)

  (define (nth n ls)
    (if (= n 0) (car ls) (nth (- n 1) (cdr ls))))

  (define (length ls)
    (if (null? ls) 0 (+ 1 (length (cdr ls)))))

  (define (assoc k ls)
    (cond [(null? ls) #f]
        [(eq? k (caar ls)) (car ls)]
        [else (assoc k (cdr ls))]))

  (define (member x ls)
    (cond [(null? ls) #f]
          [(eq? (car ls) x) ls]
          [else (member x (cdr ls))]))

  (define (find fn ls)
    (cond [(null? ls) #f]
          [(fn (car ls)) (car ls)]
          [else (find fn (cdr ls))]))

  (define (foldl fn init ls)
    (if (null? ls)
        init
        (foldl (fn (car ls) init) (cdr ls))))


  (define (foldr fn init ls)
    (if (null? ls)
        init
        (fn (car ls) (foldr fn init (cdr ls)))))

  (define (for-each fn ls)
    (if (null? ls)
        #t
        (begin (fn (car ls)) (for-each fn (cdr ls)))))

  (define (filter fn ls)
    (cond [(null? ls) '()]
          [(fn (car ls)) (cons (car ls) (filter fn (cdr ls)))]
          [else (filter fn (cdr ls))]))

  (define (map fn ls)
    (if (pair? ls)
        (cons (fn (car ls)) (map fn (cdr ls)))
        '()))

  (define (max ls)
    (cond ((null? ls) 0)
          ((null? (cdr ls) (car ls)))
          (else (let ([l (cdr ls)]
                      [r (max (cdr ls))])
                  (if (> () r) l r)))))

  (define-const = eq?)
  )
```

Listing C.2: Standard library, included in all source files. It contains general-purpose function definitions.

C.3 Macro definitions

```
(ssmodule macros
  (define-macro define :
    (define (proc1 param1 param2 . param3) body ...)
    =>
    (define proc1 (lambda (param1 param2 . param3) body ...))
    )

  (define-macro begin ::
    (begin (print a) b)
    =>
    (let () (print a) b))

  (define-macro when ::)
  (define-macro unless ::)
```

```

(define-macro let ::
  (let ([a (proc1 1 2)]
        [b (proc2 3 4)])
    body ...)
  =>
  ((lambda (a b) body ...)
   (proc1 1 2) (proc2 3 4)))

(define-macro let* ::
  (let* ([a (proc1 1 2)]
         [b (proc2 a)])
    body ...)
  =>
  (let ([a (proc1 1 2)])
    (let ([b (proc2 a)])
      body ...)))

(define-macro letrec ::
  (letrec ([recproc (lambda (x) (recproc x))])
    (recproc 1))
  =>
  (let ([recproc ()])
    (set! recproc (lambda (x) (recproc x)))
    (recproc 1)))

(define-macro cond ::
  (cond [(null? x) x]
        [(pair? x) (print x) (car x)]
        [else (error)])
  =>
  (if (null? x) x
      (if (pair? x) (begin (print x) (car x))
          (error))))

(define-macro case ::
  (case var
    [(a b c) 1]
    [(d) 2]
    [else #f])
  =>
  (let ([v var])
    (if (member v '(a b c)) 1
        (if (eq? v 'd) 2 #f))))

(define-macro and ::
  (and a b c)
  =>
  (if a (if b c)))

(define-macro or ::
  (or a b c)
  =>
  (let ([v1 a])
    (if v1 v1
        (let ([v2 b])
          (if v2 v2 c)))))

(define-macro define-handler ::
  (define-handler (sample-msg val1 val2 val3) body ...)
  =>
  (include sample-msg)
  (define (sample-msg (src val1 al2 val3) body ...)))

(define-macro define-event ::
  (define-event (user-button on?)
    (if on? (blink 7)
        (blink 0)))
  =>
  (define (user-button-loop on?)
    (if on?
        (blink 7)
        (blink 0))
    (user-button-loop (user-button)))
  (user-button-loop (user-button)))

(define-macro msg ::
  (msg sample-msg val1 val2 val3)
  =>
  (list 'sample-msg val1 val2 val3))
)
```

Listing C.3: Definitions of all macros used throughout this work. Macros are defined here as pseudo-code showing examples of use and expansion result of the macro.

C.4 Collection protocol definitions

```
(ssmodule collection

  (provide
   send-root send-parent recv-parent etx parent neighbors neighbors/quality)

  (define-primitive send-root (sender CollectSender))
  (define-primitive send-parent (sender InterceptSender))
  (define-primitive recv-parent (receiver InterceptReceiver))


  (define-primitive etx (simple EtxPrim))
  (define-primitive parent (simple ParentPrim))
  (define-primitive hops-to-root (simple HopsPrim))
  (define-primitive neighbors (simple NeighborsPrim))
  (define-primitive neighbors/quality (simple NeighQualityPrim))

  )
```

Listing C.4: Definitions of primitives used to make available the collection tree protocol to SensorScheme programs.

Acknowledgements

At the end of this long journey of that finished with writing this dissertation, it is time to thank all who have played a role in this process.

First and foremost, I want to thank my promotor and supervisor, Prof. Paul Havinga, for making it possible to undertake this project. He has provided the environment and financial means necessary and given me great freedom to find my own direction.

Second, I want to thank Dr. Jan Kuper, who has grown to become my daily supervisor, for the countless hours of fun and inspirational conversations and advice, as well as the heaps of critical commentary and fiery discussions. You have really helped to make this thesis what it is and I could not have ended up here without you.

The years in which I have been working towards this I have had many colleagues in the Pervasive Systems group and elsewhere at the UT who have somehow contributed to and made the experience more pleasurable.

Postdocs Nirvana Meratnia and Maria Lijding have played a significant role: as co-authors and in advising and talking through many of the topics that did and did not end up in this thesis.

My roommates for over three years, Kavitha, Ozlem, Mihai and Raluca; Roland Gemesi, Sinan Kaya and Hilbrand Baarsma, who have not managed to finish their promotion project, but I greatly enjoyed your company while you were around.

The many lunches these years were a truly fun experience thanks to the group of colleagues regularly joining: Berend-Jan van der Zwaag, Hans Scholten, Pierre Jansen, Hylke van Dijk, Nirvana, Maria and others from time to time. I enjoyed the many long and interesting conversations about technology, politics and the state of the world in general...

Rom Langerak, Marielle Stoelinga and Wouter Everse worked with me on one of the topics that did not end up in this thesis – a correctness proof for a

WSN routing protocol – but the experience was very educational for me, and above all, we had a nice time and a good collaboration.

I have very much enjoyed the company of new colleagues Bram Dil, Ardjan Zwartjes and Marlies van de Voort. Bram and I supervised the 2008 Codesign practicum and later joined the group as an AIO, Ardjan has been my roommate for about a year, and I enjoyed many trainrides home with Marlies. I thank you all for the many enjoyable hours spent, and friendship grown from it.

Last but not least, I need to express my great gratitude to my family for their support these years, both practical, in giving me the time needed to finish the last months, as well as the emotional support I received. I could not have done this without you.

These last five years have been brought a lot of familial changes, both good and bad, making them truly unforgettable. This it the time I got married to my wife Winda, and my two children Senno and Reza were born. Coming home to you every day has given me so much love and and pleasure in life, and your support gave me the motivation to continue when it was hardest.

I'm also very grateful for the presence and support of my parents and brothers and sisters, and a special thanks to my father Herman, who I could always turn to for advice, inspiration and to let off the occasional frustration.

My sister Sonja deserves special mention and thanks for designing the book cover and helping me with the pictures inside, and for being a great neighbor the months we lived so nearby.

I cannot finish these acknowledgements without taking time to remember Viola my sister who has been a member of my core family, living with us and sleeping over so frequently for over one and a half years and then so suddenly left this world. I am very happy to have had this very special time with you.

My memory also goes out to Hartono, my father in-law, who died recently. I haven't gotten to know you as well as I had hoped. Your loss has truly struck me and the family, and we miss you.

Bibliography

[ABC$^+$04] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George-and S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S.Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. *icdcs*, 00:582–589, 2004.

[ADH$^+$98] H. Abelson, R. K. Dybvig, C. T. Haynes, G. J. Rozas, N. I. Adams IV, D. P. Friedman, E. Kohlbecker, Jr. G. L. Steele, D. H.Bartley, R. Halstead, D. Oxley, G. J. Sussman, G. Brooks, C. Hanson, K. M. Pitman, and M. Wand. Revised^5 report on the algorithmic language scheme. *Higher Order Symbol. Comput.*, 11(1):7–105, 1998.

[ARE$^+$05] Anish Arora, Rajiv Ramnath, Emre Ertin, Prasun Sinha, Sandip Bapat, Vinayak Naik, Vinod Kulathumani, Hongwei Zhang, Hui Cao, Mukundan Sridharan, Santosh Kumar, Nick Seddon, Chris Anderson, Ted Herman, Nishank Trivedi, Chen Zhang, Mikhail Nesterenko, Romil Shah, Sandeep Kulkarni, Mahesh Aramugam, Limin Wang, Mohamed Gouda, Young ri Choi, David Culler, Prabal Dutta, Cory Sharp, Gilman Tolle, Mike Grimmer, Bill Ferriera, and Ken Parker. Exscal: Elements of an extreme scale wireless sensor network. *Real-Time Computing Systems and Applications, International Workshop on*, 0:102–108, 2005.

[ASS96] Harold Abelson, Gerald Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press, July 1996.

[AY05] Kemal Akkaya and Mohamed Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3:325–349, 2005.

[AYKC04] Junaid Ahsen Ali, Won-Sik Yoon, Jai-Hoon Kim, and We-Duke Cho. U-kitchen: Application scenario. *Software Technologies for Future Embedded and Ubiquitous Systems, IEEE Workshop on*, 0:169, 2004.

[BBKO05] A. Bestavros, A.D. Bradley, A.J. Kfoury, and M.J. Ocean. Snbench: a development and run-time platform for rapid deployment of sensor network applications. *Broadband Networks, 2005 2nd International Conference on*, pages 957–966 Vol. 2, Oct. 2005.

[BCD$^+$05] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[BGL$^+$07] Matthew Brown, Seth Gilbert, Nancy Lynch, Calvin Newport, Tina Nolte, and Michael Spindel. The virtual node layer: a programming abstraction for wireless sensor networks. *SIGBED Rev.*, 4(3):7–12, 2007.

[BGS01] Philippe Bonnet, Johannes Gehrke, and Praveen Seshadri. Towards sensor database systems. In *MDM '01: Proceedings of the Second International Conference on Mobile DataManagement*, pages 3–14, London, UK, 2001. Springer-Verlag.

[Bha01] Pravin Bhagwat. Bluetooth: Technology for short-range wireless apps. *IEEE Internet Computing*, 5(3):96–103, 2001.

[BHS03] Athanassios Boulis, Chih-Chieh Han, and Mani B. Srivastava. Design and implementation of a framework for efficient and programmablesensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems,applications and services*, pages 187–200, New York, NY, USA, 2003. ACM Press.

[BK06] Urs Bischoff and Gerd Kortuem. Rulecaster: A macroprogramming system for sensor networks. In *Proceedings OOPSLA Workshop on Building Software for Sensor Networks*, 2006.

[BKM$^+$04] Jan Beutel, Oliver Kasten, Friedemann Mattern, Kay Römer, Frank Siegemund, and Lothar Thiele. Prototyping wireless sensor network applications with btnodes. In *1st European Workshop on Wireless Sensor Networks (EWSN)*, number 2920 in LNCS, pages 323–338, Berlin, Germany, January 2004. Springer-Verlag.

[BKZD04] Michael Beigl, Albert Krohn, Tobias Zimmer, and Christian Decker. Typical sensors needed in ubiquitous and pervasive computing. In *in Proceedings of the First International Workshop on Networked Sensing Systems (INSS '04*, pages 153–158, 2004.

[BLC09] Niels Brouwers, Koen Langendoen, and Peter Corke. Darjeeling, a feature-rich vm for the resource poor. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 169–182, New York, NY, USA, 2009. ACM.

[Bon91] A. Bondorf. Similix manual, system version 4.0. Technical report, DIKU, Department of Computer Science, University of Copenhagen, 1991.

[BPRL05] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, Berkeley, CA, USA, 2005. USENIX Association.

[btn09] Btnut system software. `http://www.btnode.ethz.ch/static_docs/doxygen/btnut/`, January 2009.

[Bur02] Alexander Burger. The pico lisp reference. `http://www.software-lab.de/ref.html`, dec 2002.

[CHK$^+$04] W. Steven Conner, John Heidemann, Lakshman Krishnamurthy, Xi Wang, and Mark Yarvis. Workplace applications of sensor networks. Technical report, USC/Information Sciences Institute, 2004.

[Con93] C. Consel. A tour of schism: A partial evaluation system for higher-order applicative languages. In *Proceedings of the*

1993 ACM SIGPLAN symposium on Partial evaluation and semantics-based program manipulation, pages 145–154, 1993.

[Cor08] Sentilla Corporation. Sentilla perk: Pervasive computing kit. `http://www.sentilla.com/perkpage.html`, 2008.

[Croa] Crossbow Technology. High-performance wireless sensor network node. Data Sheet Document Part Number: 6020-0117-02 Rev A, Crossbow Technology.

[Crob] Crossbow Technology. Mica2 wireless measurement system. Data Sheet Document Part Number: 6020-0042-08 Rev A, Crossbow Technology.

[Croc] Crossbow Technology. Stargate netbridge. Data Sheet Document Part Number: 6020-0126-01 Rev A, Crossbow Technology.

[Cro03] Crossbow Technology. Mote in-network programming user reference, 2003. `http://www.tinyos.net/tinyos-1.x/doc/NetworkReprogramming.pdf`.

[CSC06] Li-Der Chou, Chun-Cheng Sheu, and Home-Way Chen. Design and prototype implementation of a novel automatic vehicle parking system. In *ICHIT '06: Proceedings of the 2006 International Conference on Hybrid Information Technology*, pages 292–297, Washington, DC, USA, 2006. IEEE Computer Society.

[DB72] N. G. De Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae*, pages 381–392, 1972.

[DBK$^+$04] Christian Decker, Michael Beigl, Albert Krohn, Philip Robinson, and Uwe Kubach. eseal - a system for enhanced electronic assertion of authenticity and integrity. In Alois Ferscha and Friedemann Mattern, editors, *Pervasive*, volume 3001 of *Lecture Notes in Computer Science*, pages 254–268. Springer, 2004.

[DF05] Danny Dubé and Marc Feeley. Bit: A very compact scheme system for microcontrollers. *Higher Order Symbol. Comput.*, 18(3-4):271–298, 2005.

[DFEV06] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, November 2006.

[DGV04] Adam Dunkels, Björn Grönvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors (Emnets-I)*, Tampa, Florida, USA, November 2004.

[DHB93] R. Kent Dybvig, Robert Hieb, and Carl Bruggeman. Syntactic abstraction in Scheme. *Lisp and Symbolic Computation*, 5(4):295–326, 1993.

[Dic92] Ken Dickey. Scheming with objects. *AI Expert*, (7(10)):24–33, October 1992.

[DSVA06] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems (SenSys'06)*, pages 29–42, Boulder, Colorado, USA, November 2006.

[Dun03] Adam Dunkels. Full tcp/ip for 8 bit architectures. In *Proceedings of First ACM/Usenix International Conference on Mobile Systems, Applications and Services (MobiSys 2003)*, page 14, San Francisco, USA, 2003.

[Dun05] Adam Dunkels. Towards TCP/IP for Wireless Sensor Networks. Licentiate thesis, March 2005.

[Dun07] Adam Dunkels. Rime — a lightweight layered communication stack for sensor networks. In *Proceedings of the European Conference on Wireless Sensor Networks (EWSN), Poster/Demo session*, Delft, The Netherlands, January 2007.

[Dyb09] R. Kent Dybvig. *The Scheme Programming Language.* MIT Press, fourth edition, 2009.

[EBMP+05] L. Evers, M. J. J. Bijl, M. Marin-Perianu, R. Marin-Perianu, and P. J. M. Havinga. Wireless sensor networks and beyond: A case study on transport and logistics. Technical Report TR-CTIT-05-26, Univ. of Twente, Enschede, June 2005.

[EF01] William S. Evans and Christopher W. Fraser. Bytecode compression via profiled grammar rewriting. In *PLDI '01: Proceedings of the ACM SIGPLAN 2001 conference on Programming language design and implementation*, pages 148–155, New York, NY, USA, 2001. ACM.

[EHK07a] L. Evers, P. J. M. Havinga, and J. Kuper. Dynamic sensor network reprogramming using sensorscheme. In *Proceedings of the 18th Annual IEEE Symposium on Personal, Indoor and Mobile Radio Communications, Athens, Greece*, pages 1–5, Los Alamitos, September 2007. IEEE Computer Society Press.

[EHK07b] L. Evers, P. J. M. Havinga, and J. Kuper. Flexible sensor network reprogramming for logistics. Technical Report TR-CTIT-07-51, Enschede, July 2007.

[EHK07c] L. Evers, P. J. M. Havinga, and J. Kuper. Flexible sensor network reprogramming for logistics. In *Proceedings of the Fourth IEEE International Conference on Mobile Ad-hoc and Sensor Systems, MASS 2007, Pisa, Italy*, Piscataway, October 2007. IEEE Computer Society Press.

[EHK+07d] L. Evers, P. J. M. Havinga, J. Kuper, M. E. M. Lijding, and N. Meratnia. Sensorscheme: Supply chain management automation using wireless sensor networks. In *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2007, Patras, Greece*, pages 448–455, Los Alamitos, September 2007. IEEE Computer Society Press.

[EHK+07e] L. Evers, P. J. M. Havinga, J. Kuper, M. E. M. Lijding, and N. Meratnia. Sensorscheme: Supply chain management automation using wireless sensor networks. In *Proceedings of the 12th IEEE Conference on Emerging Technologies and Factory Automation, ETFA 2007, Patras, Greece*, pages 448–455, Los Alamitos, September 2007. IEEE Computer Society Press.

[EK09] L. Evers and J. Kuper. Partially evaluated sensor networks: Automatic specialization for heterogeneous wireless sensor & actuator networks. In *Proceedings of the 2009 ACM SIGPLAN workshop on Partial evaluation and program manipulation, Savannah, GA, USA*, pages 73–80, New York, January 2009. ACM.

[ELK08] L. Evers, M. E. M. Lijding, and J. Kuper. Generic multi-packet communication through object serialization. In *Proceedings of the 3rd international workshop on Middleware for sensor networks, Leuven, Belgium*, pages 25–30, New York, December 2008. ACM.

[FC08] Aurélien Francillon and Claude Castelluccia. Code injection attacks on harvard-architecture devices. In *CCS '08: Proceedings of the 15th ACM conference on Computer and communications security*, pages 15–26, New York, NY, USA, 2008. ACM.

[FD01] Darrell Ferguson and Dwight Deugo. Call with current continuation patterns. In *8th Conference on Pattern Languages of Programs*, September 2001.

[FD03] Marc Feeley and Danny Dubé. Picbit: A scheme system for the pic microcontroller. In *Proceedings of the Fourth Workshop on Scheme and Functional Programming, Boston, Massachusetts, USA*, November 2003.

[Fel85] Matthias Felleisen. Transliterating prolog into scheme. Technical Report Computer Science Technical Report 182, Indiana University, October 1985.

[Fir] FireBug. `http://firebug.sourceforge.net`.

[FRL05] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Mobile agent middleware for sensor networks: an application case study. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 51, Piscataway, NJ, USA, 2005. IEEE Press.

[FS09] Matthew Flatt and PLT Scheme. Reference: PLT scheme. Reference Manual PLT-TR2009-reference-v4.2, PLT Scheme Inc., June 2009. `http://plt-scheme.org/techreports/`.

[Gat00] Erann Gat. Lisp as an alternative to java. *Intelligence*, 11:2000, 2000.

[GBCT06] Chandana Gamage, Kemal Bicakci, Bruno Crispo, and Andrew S. Tanenbaum. Security for the mythical air-dropped sensor network. In *ISCC '06: Proceedings of the 11th IEEE Symposium on Computers and Communications*, pages 41–47, Washington, DC, USA, 2006. IEEE Computer Society.

[GJP$^+$06] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM.

[GKGM05] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *Proceedings of the twentieth ACM symposium on Operating systems principles*, 2005.

[GKW$^+$02] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. Technical Report RB-TR-02-003, Intel Research, march 2002.

[GLvB$^+$03] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D.Culler. The nesc language: A holistic approach to networked embedded systems, 2003.

[HC02] Jason L. Hill and David E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[HC04] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM.

[HC08] Jonathan W. Hui and David E. Culler. Extending ip to low-power, wireless personal area networks. *IEEE Internet Computing*, 12(4):37–45, 2008.

[HHKK04] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004.

[Hil03] Jason Hill. Spec takes the next step toward the vision of true smart dust. `http://www.jlhlabs.com/jhill_cs/spec/`, March 2003.

[HKS+04] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Jonathan Hui, and Bruce Krogh. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 270–283, New York, NY, USA, 2004. ACM.

[HKS+05] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems,applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[HKW85] F. Hattori, K. Kushima, and T. Wasano. A comparison of lisp, prolog, and ada programming productivity in ai area. In *ICSE '85: Proceedings of the 8th international conference on Software engineering*, pages 285–291, Los Alamitos, CA, USA, 1985. IEEE Computer Society Press.

[Hol00] Seth Edward-Austin Hollar. Cots dust. Master's thesis, University of California, Berkeley, 2000.

[HSW+00] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David E.Culler, and Kristofer S. J. Pister. System architecture directions for networked sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[IEE06] IEEE Standard. *Wireless medium access control and physical layer specifications for low-rate wireless personal area networks.*, 802.15.4-2006 edition, september 2006.

[Int] Intel Corporation. The wireless vineyard. `http://www.intel.com/technology/techresearch/research/rs01031.htm`.

[JC04] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *First IEEE Comm. Soc. Conf. on Sensor and Ad Hoc Communications and Networks*, 2004.

[JDK$^+$05] Professor James, W. Demmel, Sukun Kim, Sukun Kim, and Sukun Kim. Wireless sensor networks for structural health monitoring. Technical report, in UC Berkeley Master's Thesis, 2005.

[JGS93] N.D. Jones, C.K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* 1993.

[Jon03] Simon P. Jones. *Haskell 98 Language and Libraries: The Revised Report.* Cambridge University Press, May 2003.

[JOW$^+$02] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet, 2002.

[JSS89] N.D. Jones, P. Sestoft, and H. Søndergaard. Mix: A self-applicable partial evaluator for experiments in compiler generation. *Lisp and Symbolic Computation*, 2(1):9–50, 1989.

[JSS00] Chaiporn Jaikaeo, Chavalit Srisathapornphat, and Chien-Chung Shen. Querying and Tasking in Sensor Networks. In *SPIE's 14th Annual International Symposium on Aerospace/Defense Sensing, Simulation, and Control (Digitization of the Battlespace V)*, Orlando, Florida, April 24–28 2000.

[KC08] Marcin Karpiński and Vinny Cahill. Stream-based macro-programming of wireless sensor, actuator network applications with sosna. In *DMSN '08: Proceedings of the 5th workshop on Data management for sensor networks*, pages 49–55, New York, NY, USA, 2008. ACM.

[KFG$^+$03] R. Kremens, J. Faulring, A. Gallagher, A. Seema, and A. Vodacek. Autonomous field-deployable wildland fire sensors. *International Journal of Wildland Fire*, 12(2):237–244, June 2003.

[KGMG07] Nupur Kothari, Ramakrishna Gummadi, Todd Millstein, and Ramesh Govindan. Reliable and efficient programming abstractions for wireless sensor networks. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language* *design and implementation*, pages 200–210, New York, NY, USA, 2007. ACM.

[Kis99] Oleg Kiselyov. Purely-functional object-oriented system. `http://okmij.org/ftp/Scheme/oop-in-fp.txt`, December 1999.

[Kno04] Terry Knott. Smart surrogates. *BP Frontiers magazine*, Issue 9:6–10, april 2004.

[KOA$^+$99] Cory D. Kidd, Robert Orr, Gregory D. Abowd, Christopher G. Atkeson, Irfan A. Essa, Blair MacIntyre, Elizabeth D. Mynatt, Thad Starner, and Wendy Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *CoBuild '99: Proceedings of the Second International Workshop on Cooperative Buildings, Integrating Information, Organization, and Architecture*, pages 191–198, London, UK, 1999. Springer-Verlag.

[KR05] Oliver Kasten and Kay Römer. Beyond event handlers: programming wireless sensors with attributed state machines. In *IPSN*, pages 45–52, 2005.

[Kri02] Shriram Krishnamurthi. How to design programs: a new look at introductory computing. *J. Comput. Small Coll.*, 17(6):4–5, 2002.

[KSMA06] YoungMin Kwon, Sameer Sundresh, Kirill Mechitov, and Gul Agha. Actornet: an actor platform for wireless sensor networks. In *AAMAS '06: Proceedings of the fifth international joint conference on Autonomous agents and multiagent systems*, pages 1297–1300, New York, NY, USA, 2006. ACM.

[KW05] Sandeep S. Kulkarni and Limin Wang. Mnp: Multihop network reprogramming service for sensor networks. *Distributed Computing Systems, International Conference on*, 0:7–16, 2005.

[Lan07] K. Langendoen. Medium access control in wireless sensor networks. In H. Wu and Y. Pan, editors, *Medium Access Control in Wireless Networks, Volume II: Practice and Standards.* Nova Science Publishers, Inc., 2007.

[Lat00] Mario Latendresse. Automatic generation of compact programs and virtual machines for scheme. In *Proceedings of the Workshop on Scheme and Functional Programming*, pages 45–52, 2000.

[LBV06] K Langendoen, A Baggio, and O Visser. Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *In WPDRTS 2006*, 2006.

[LC02] Philip Levis and David Culler. Mate: A tiny virtual machine for sensor networks, 2002.

[LCL$^+$03] Jie Liu, Maurice Chu, Juan Liu, James Reich, and Feng Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 2(4):50–62, 2003.

[LFO$^+$07] Joshua Lifton, Mark Feldmeier, Yasuhiro Ono, Cameron Lewis, and Joseph A. Paradiso. A platform for ubiquitous sensor deployment in occupational and domestic environments. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 119–127, New York, NY, USA, 2007. ACM.

[LGC04] P. Levis, D. Gay, and D. Culler. Bridging the gap: Programming sensor networks with application specific virtual machines. Technical Report CSD-04-1343, UC Berkeley, Aug 2004.

[LGC05] P. Levis, D. Gay, and D. Culler. Active sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design andImplementation (NSDI)*, May 2005.

[LMG$^+$04] Philip Levis, Sam Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric Brewer, and David Culler. The emergence of networking abstractions and techniques in tinyos. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 1–1, Berkeley, CA, USA, 2004. USENIX Association.

[LN79] Hugh C. Lauer and Roger M. Needham. On the duality of operating system structures. *SIGOPS Oper. Syst. Rev.*, 13(2):3–19, 1979.

[LPCS04] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *In Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI*, pages 15–28, 2004.

[LRST07] Clemens Lombriser, Daniel Roggen, Mathias Stäger, and Gerhard Tröster. Titan: A tiny task network for dynamically reconfigurable heterogeneous sensor networks. In *15. Fachtagung Kommunikation in Verteilten Systemen (KiVS)*, Informatik aktuell, pages 127–138. Springer, 0 2007.

[LRZ03] Juan Liu, James Reich, and Feng Zhao. Collaborative in-network processing for target tracking. *EURASIP J. Appl. Signal Process.*, 2003:378–391, 2003.

[LS05] Dimitrios Lymberopoulos and Andreas Savvides. Xyz: a motion-enabled, power aware sensor node platform for distributed sensor network applications. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 63, Piscataway, NJ, USA, 2005. IEEE Press.

[LY99] Tim Lindholm and Frank Yellin. *The Java(TM) Virtual Machine Specification (2nd Edition).* Prentice Hall PTR, April 1999.

[MAK07] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 41(3):145–158, 2007.

[May04] K. Mayer. Instrumenting cattle – real time health monitoring of cattle using wirelesstechnologies. Poster for Sir Mark Oliphant Conference 2004 "Converging Technologies forAgriculture and Environment", August 2004.

[McC60] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part i. *Commun. ACM*, 3(4):184–195, 1960.

[McC62] John McCarthy. *LISP 1.5 Programmer's Manual.* The MIT Press, 1962.

[MCPA02] Alan Mainwaring, David Culler, Joseph Polastre, and Robert Szewczykand John Anderson. Wireless sensor networks for habitat monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networksand applications*, pages 88–97. ACM Press, 2002.

[MFHW02] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and WeiHong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36, 2002.

[MFHW03] Samuel Madden, Michael J. Franklin, Joseph M. Hellerstein, and WeiHong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conferenceon Management of data*, pages 491–502, New York, NY, USA, 2003. ACM Press.

[MGL$^+$06] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN2006)*, pages 212–227, February 2006.

[MLM$^+$05] P.J. Marron, A. Lachenmann, D. Minder, J. Hahner, R. Sauter, and K. Rothermel. Tinycubus: a flexible and adaptive framework sensor networks. *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 278–289, Jan.-2 Feb. 2005.

[MMWN07] Geoffrey Mainland, Greg Morrisett, Matt Welsh, and Ryan Newton. Sensor network programming with flask. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 385–386, New York, NY, USA, 2007. ACM.

[MnWHG$^+$05] A. Mohan, null Wei Hong, D. Gay, P. Buonadonna, and A. Mainwaring. End-to-end performance characterization of sensornet multi-hop routing. *International Conference on Pervasive Services*, 0:27–36, 2005.

[Mot06] Moteiv. *Tmote Sky Datasheet http://www.sentilla.com/pdf/eol/tmote-sky-datasheet.pdf*, 2006.

[MP04] DANIEL MACHALABA and ANDY PASZTOR. Thinking inside the box: Shipping containers get 'smart'. *The Wall Street Joural online*, January 15 2004.

[MP06] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks with logical neighborhoods. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 8, New York, NY, USA, 2006. ACM.

[MRD$^+$07] Rene Mueller, Jan S. Rellermeyer, Michael Duller, Gustavo Alonso, and Donald Kossmann. A dynamic and flexible sensor network platform. In *SIGMOD '07: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*, pages 1085–1087, New York, NY, USA, 2007. ACM.

[MS06] William P. McCartney and Nigamanth Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, New York, NY, USA, 2006. ACM.

[nan] The nanovm - java for the avr. `http://www.harbaum.org/till/nanovm/index.shtml`.

[Neta] Network Working Group. Tep 116: Packet protocols.

[Netb] Network Working Group. Tep 123: Collection tree protocol (ctp).

[NMW07] Ryan Newton, Greg Morrisett, and Matt Welsh. The regiment macroprogramming system. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 489–498, New York, NY, USA, 2007. ACM.

[NMZ$^+$05] Leyla Nazhandali, Michael Minuth, Bo Zhai, Javin Olson, Todd Austin, and David Blaauw. A second-generation sensor network processor with application-driven memory optimizations and out-of-order execution. In *CASES '05: Proceedings of the 2005 international conference on Compilers, architectures and synthesis for embedded systems*, pages 249–256, New York, NY, USA, 2005. ACM.

[NPR06] Christopher Nitta, Raju Pandey, and Yann Ramin. Y-threads: Supporting concurrency in wireless sensor networks. In Phillip B. Gibbons, Tarek F. Abdelzaher, James Aspnes, and Ramesh Rao, editors, *DCOSS*, volume 4026 of *Lecture Notes in Computer Science*, pages 169–184. Springer, 2006.

[NW04] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data managementfor sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[OCa] Objective caml (ocaml) programming language website. http://caml.inria.fr/.

[Pow95] R.A. Powers. Batteries for low power electronics. *Proceedings of the IEEE*, 83(4):687–693, Apr 1995.

[Pre00] Lutz Prechelt. An empirical comparison of c, c++, java, perl, python, rexx, and tcl, 2000.

[Pro09] Project Sun SPOT. Sun spot world. `http://www.sunspotworld.com`, Oktober 2009 2009.

[PSC05] Joseph Polastre, Robert Szewczyk, and David E. Culler. Telos: enabling ultra-low power wireless research. In *IPSN*, pages 364–369, 2005.

[RL03] Niels Reijers and Koen Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM.

[RM04] Kay Römer and Friedemann Mattern. The design space of wireless sensor networks, December 2004.

[Röm04] Kay Römer. Tracking real-world phenomena with smart dust. In Holger Karl, Andreas Willig, and Adam Wolisz, editors, *EWSN*, volume 2920 of *Lecture Notes in Computer Science*, pages 28–43. Springer, 2004.

[Ruf93] E. Ruf. *Topics in Online Partial Evaluation.* PhD thesis, Stanford University, California, February 1993. Published as technical report CSL-TR-93-563.

[SFCB05] J. Steffan, L. Fiege, M. Cilia, and A. Buchmann. Towards multipurpose wireless sensor networks, 2005.

[SGKK04] Martin Strohbach, Hans Gellersen, Gerd Kortuem, and Christian Kray. Cooperative artefacts: Assessing real world situations with embedded technology. In *Proceedings of the International Conference on Ubiquitous Computing (Ubicomp)*, pages 250–267. Springer, Berlin, Heidelberg, New York, 2004.

[SHE03] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, University of California, Los Angeles, Center for Embedded Networked Computing, November 2003.

[SHW08] Tamim I. Sookoor, Timothy W. Hnat, and Kamin Whitehouse. Programming cyber-physical systems with macrolab. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 363–364, New York, NY, USA, 2008. ACM.

[SKGK04] Martin Strohbach, Gerd Kortuem, Hans-Werner Gellersen, and Christian Kray. Using cooperative artefacts as basis for activity recognition. In Panos Markopoulos, Berry Eggen, Emile H. L. Aarts, and James L. Crowley, editors, *EUSAI*, volume 3295 of *Lecture Notes in Computer Science*, pages 49–60. Springer, 2004.

[SLO05] L. Szumel, J. LeBrun, and J. D. Owens. Towards a mobile agent framework for sensor networks. In *EmNets '05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors*, pages 79–87, Washington, DC, USA, 2005. IEEE Computer Society.

[Sou] SourceForge.net repository of TinyOS. `http://tinyos.cvs.sourceforge.net/viewvc/tinyos/`.

[sou09] Open source. The computer language benchmarks game. Web site, 2009.

[Sta02] V. Stanford. Using pervasive computing to deliver elder care. *Pervasive Computing, IEEE*, 1(1):10–13, Jan-Mar 2002.

[SW67] H. Schorr and W. M. Waite. An efficient machine-independent procedure for garbage collection in various list structures. *Commun. ACM*, 10(8):501–506, 1967.

[Tec] Crossbow Technology. Micaz wireless measurement system. Data Sheet Document Part Number: 6020-0060-04 Rev A, Crossbow Technology.

[The03] The Ohio State University NEST team. A Line in the Sand: A DARPA-NEST Field Experiment. `http://www.cse.ohio-state.edu/siefast/nest/nest_webpage/ALineInTheSand.html`, August 2003.

[vBCB03] Rob von Behren, Jeremy Condit, and Eric Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[VMKP03] Elena Vildjiounaite, Esko-Juhani Malm, Jouni Kaartinen, and PetteriAlahuhta. Networking of smart things in a smart home. In *UBIHCISYS 2003 Online Proceedings*, 2003.

[VR09] Peter Van Roy. Programming paradigms for dummies: What every programmer should know. In G. Assayag and A. Gerzso, editors, *New Computational Paradigms for Computer Music*. IRCAM/Delatour, France, 2009.

[WAJR$^+$05] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 108–120, Jan.-2 Feb. 2005.

[WDLS06] Chonggang Wang, Mahmoud Daneshmand, Bo Li, and Kazem Sohraby. A survey of transport protocols for wireless sensor networks. *IEEE Network Magazine Special Issue on Wireless Sensor Networking*, 20(Issue 3):34 – 40, May-June 2006.

[Wei96] Mark Weiser. Ubiquitous computing. `http://nano.xerox.com/hypertext/weiser/UbiHome.html`, March 1996.

[WJNB95] Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. Dynamic storage allocation: A survey and critical review. pages 1–116. Springer-Verlag, 1995.

[WLLP01] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[WM04] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *First USENIX/ACM Symposium on Networked Systems Design and Implementation(NSDI '04)*, March 2004.

[WO05] Lars Wirzenius and Kenneth Oksanen. Hedgehog: A tiny lisp for embedded applications. Technical report, February 2005.

[WSBC04] Kamin Whitehouse, Cory Sharp, Eric Brewer, and David Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems,applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM Press.

[WSGLA08] Zheng Wang, Hamid Sadjadpour, and Jose Joaquin Garcia-Luna-Aceves. The capacity and energy efficiency of wireless ad hoc networks with multi-packet reception. In *MobiHoc '08: Proceedings of the 9th ACM international symposium on Mobile ad hoc networking and computing*, pages 179–188, New York, NY, USA, 2008. ACM.

[WZL06] Kamin Whitehouse, Feng Zhao, and Jie Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Wireless Sensor Networks*, pages 5–20. 2006.

[Yan03] S. Yang. Redwoods go high tech: Researchers use wireless sensors to study California'sstate tree. Press release, University of California, Berkeley, July 2003.

[YS03] H. Yang and B. Sikdar. A protocol for tracking mobile targets using sensor networks. *Sensor Network Protocols and Applications,* *2003. Proceedings of the First IEEE. 2003 IEEE International Workshop on*, pages 71–81, May 2003.

[ZCH07] Y. Zhang, S. Chatterjea, and P. J. M. Havinga. Experiences with implementing a distributed and self-organizing scheduling algorithm for energy-efficient data gathering on a real-life sensor network platform. In *First IEEE International Workshop on From Theory to Practice in Wireless Sensor Networks, Helsinki, Finland*, Helsinki, Finland, June 2007. IEEE Computer Society.

[ZL77] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory*, 23:337–343, 1977.

[ZSLM04] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. Hardware design experiences in ZebraNet. In *2nd International Conference on Embedded Networked Sensor Systems*, pages 227–238. ACM Press, 2004.