UNIVERSITY OF OSLO
Department of Informatics

Programming Wireless Sensor Networks: From Static to Adaptive Models

PhD Thesis

Amirhosein
Taherkordi

Series of dissertations submitted to the
Faculty of Mathematics and Natural Sciences, University of Oslo
No. 1114

ISSN 1501-7710

Cover: Inger Sandved Anfinsen.
Printed in Norway: AIT Oslo AS.

Produced in co-operation with Unipub.
The thesis is produced by Unipub merely in connection with the
thesis defence. Kindly direct all inquiries regarding the thesis to the copyright
holder or the unit which grants the doctorate.

To my family

Abstract

Wireless Sensor Networks (WSNs) are a rapidly emerging research area because of their vast application vistas in real-world environments, as well as their rapid deployments at low cost and with high flexibility. In 2003, Technology Review ranked WSNs among 10 emerging technologies that will change the world. WSNs consist of tiny sensor nodes that can be easily embedded in the environment, establish a wireless ad-hoc network, and compose a distributed system to collaboratively sense physical phenomena and process sensed data, or to react to the environment based on the sensed data. To practically use this technology, WSNs must be able to operate unattended for long periods of time, especially when deployed in inaccessible places. Moreover, their new applications in heterogeneous and ubiquitous settings make the autonomy of their operations very important. This introduces several new requirements, such as reconfiguration of WSNs to meet future unpredictable needs, remote maintenance of sensor software, adapting WSN functionality against changes in heterogeneous environments, and remote patching of sensor software to handle after-deployment faults.

To address these requirements, we need to study the fundamental issue of *reprogramming and software reconfiguration* in WSNs and devise a framework that provides the primitives required to enable dynamicity in sensor software. This thesis focuses on this issue and presents a set of WSN programming frameworks that simplify application development in a range of settings, from static deployments with pre-defined and constant conditions to dynamic deployments with changing and unpredictable requirements. In particular, the contributions of this thesis are mainly within the following four areas.

The first part presents a distributed middleware system, called WiSeKit, to enable adaptation and reconfiguration of WSN applications in ubiquitous and context-aware environments. WiSeKit proposes a middleware software framework that formulates the process of adaptive WSN application development and abstracts the underlying technological adaptation processes. The adaptation strategy is inspired by the main activities of the *feedback control loop*, including context-awareness, adaptation reasoning, and software reconfiguration. In particular, it introduces a novel context processing model to monitor various context information (*e.g.*, sensor resources and environmental changes) in WSNs. The analyzed context data is delivered to a *hierarchical adaptation reasoning* framework to make decisions about what adaptation to perform. Finally, WiSeKit pro-

poses a component-based reconfiguration approach to implement the adaptation choices.

The second part of this thesis presents a new component-based programming model for WSNs, called REMORA. This programming abstraction is proposed not only to simplify programming in WSNs, but also to address the third goal of WISEKIT—component-based reconfiguration. REMORA offers a well-structured programming paradigm that fits very well with resource limitations of WSNs. Furthermore, the special attention to event handling in REMORA makes it more practical for WSN applications, which are inherently event-driven. More importantly, the mutualism between REMORA and underlying system software promises a new direction towards separation of concerns in WSNs.

In the third part, we reconsider REMORA in order to extend it with the capability of compositional component reconfiguration and therefore meet the component-based reconfiguration requirement of WISEKIT. The dynamicity of REMORA is achieved by the principle of *in-situ reconfigurability*, referring to minimizing the overhead of component reconfiguration through a set of in-situ updating guidelines. This is achieved within REMOWARE, a run-time system leveraging on the concept of in-situ reconfigurability to allow component-based reprogramming in WSNs. New binary update preparation, code distribution, run-time linking, dynamic memory allocation and loading, and system state preservation are the main features supported by REMOWARE.

The last contribution of this thesis is dedicated to providing a software framework for WSNs that enables the development of distributed sensor services and their integration with existing IT systems. This is achieved by extending REMORA with an interaction model inspired by the *REST architectural style* in order to facilitate interoperability of sensor services with the Internet through Web service-enabled components. The provision of such framework is very important in WISEKIT when processing context information provided by heterogeneous nodes in the network.

To evaluate the different parts of the proposed framework, we always analyze our solutions from two complementary perspectives. On one hand, we quantify the programming effort in developing non-trivial reference test use-cases both using our solutions and with mainstream programming tools. On the other hand, we explore the system performance based on metrics such as network overhead, energy overhead, and resource usage.

Acknowledgments

Praise be to God, the most gracious and the most merciful. Without his blessing and guidance my accomplishment would never have been possible.

I would like to acknowledge many people who helped me during the course of this work. First, I wish to thank my PhD advisor Professor Frank Eliassen for giving me the opportunity to be part of his research group and for providing me the right balance of guidance and independence in my research. I am greatly indebted to his full support and constant encouragement and advice both in technical and non-technical matters. I would also like to thank my co-supervisor Professor Tor Skeie for his support and advice through this research, especially during the first year.

My sincere appreciation is extended to Dr. Romain Rouvoy at INRIA-Lille, for his encouragement, constructive suggestions and comments on my thesis work. During my PhD studies, as well as four months visit at INRIA I had the great fortune and honor to collaborate with his group on problems of common interests. Furthermore, I would like to thank Dr. Frédéric Loiret from the same group for his very technical comments and feedbacks during the last year of my study.

I extend my gratitude to my colleagues at Sonitor Technologies for their support and cooperation over the past few months. I am also thankful to my former and current group-mates: Dr. Mohsen Sharifi (my master's thesis advisor), Dr. Rasool Jalili (my bachelor's thesis supervisor), Majid Alkaee Taleghan, Dr. Quan Le-Trung, and Hai Ngoc Pham.

Last, but certainly not the least, I would like to acknowledge the commitment, sacrifice and support of my family, especially my parents and my wife Mahsa, who have always motivated me. In reality this thesis is partly theirs too and eventually, I hope to have a chance to return what they gave me.

Amirhosein Taherkordi
Oslo, September 9, 2011

Preface

This dissertation has been submitted to the Faculty of Mathematics and Natural Sciences at the University of Oslo in partial fulfillment of the requirements for the degree of *Philosophiae Doctor* (PhD). The studies were carried out over a period of three and half years, from September 2007 to February 2011. The research was funded by the Research Council of Norway through the project "Scalable Wireless Sensor Networks" (SWISNET), grant number 176151. My supervisors have been Professor Frank Eliassen and Professor Tor Skeie.

The SWISNET project was located at the Department of Informatics and initiated as a research effort in order to investigate how to develop hierarchical, scalable and dynamically reprogrammable WSNs, and evaluate the developed solutions through simulation studies and experiments with real WSNs running the developed technology concepts.

The senior researchers involved in the project are Professor Frank Eliassen, Professor Tor Skeie, Dr. Paal E. Engelstad and Dr. Quan Le-Trung. Eliassen has also been the project leader. The project also financed two Dr. Scient. students, Hai Ngoc Pham and Amirhosein Taherkordi. Pham focused on the networking issues in the SWISNET project. In particular, his thesis addresses developing novel solutions for energy-efficient networking in hierarchical WSNs to support multicast including node addressing schemes and reliable delivery. The project has also attracted two external senior researchers, Dr. Romain Rouvoy and Dr. Frédéric Loiret to receive their precious comments and feedbacks on modeling, programming and reprogramming resource-constrained systems.

I spent the whole period of my PhD studies at the Department of Informatics, except a four-month period in the middle of my studies for doing an internship under the supervision of Dr. Romain Rouvoy in the National Institute for Research in Computer Science and Control (INRIA), Lille, France. The overall architecture presented in this thesis was developed cooperatively by the researchers involved in the project.

Contents

List of Figures

List of Tables

Abbreviations

WSNs	Wireless Sensor Networks
QoS	Quality of Service
REST	Representational State Transfer
SOA	Service Oriented Architectures
FCL	Feedback Control Loop
TCP	Transmission Control Protocol
HTTP	Hypertext Transfer Protocol
RPC	Remote Procedure Call
SOAP	Simple Object Access Protocol
WSDL	Web Services Description Language
UPnP	Universal Plug and Play
SCA	Service Component Architecture
ELF	Executable and Linkable Format
CBSE	Component-Based Software Engineering
JVM	Java Virtual Machine
RFID	Radio-Frequency IDentification
RMI	Remote Method Invocation
OS	Operating System
IDL	Interface Definition Language
URI	Uniform Resource Identifier
HVAC	Heating, Ventilation, and Air Conditioning
JSON	JavaScript Object Notation

Part I

Overview

Chapter 1

Introduction

Advances in wireless communications and miniaturization of hardware components have enabled the development of low-cost, low-power and multifunctional sensor nodes. These devices are small in size and communicate in short distances over a radio frequency channel. These tiny nodes, which consist of sensing, data processing and communicating components, realize the objectives of sensor networks. The concept of *Wireless Sensor Networks* (WSNs) was originally proposed in an article in 1999 about "smart dust"—computers that can be sprayed on the wall, deployed anywhere throughout the environment, and collaborate to solve big problems [1]. Later on, WSNs found their way into a wide variety of applications with vastly varying requirements and characteristics. According to a widely-accepted definition, a WSN is composed of a large number of integrated sensor nodes that are densely deployed either inside a phenomenon or very close to it, and collaborate through a wireless network in collecting environmental information or reacting to specific events [2, 3, 4].

Since the main mission of WSNs is to bridge the gap between the physical world and the virtual world, applications for these types of networks are often characterized by close coupling between the physical and computing disciplines. WSNs have been developed for a wide range of applications such as habitat monitoring, object tracking, precision agriculture, building monitoring, military systems, healthcare, etc. [5, 6, 7, 8, 9, 10, 11] Whereas the earlier WSN applications were limited to a single function called "sense and send" with trivial local data processing tasks, the new emerging applications for WSNs are gradually moving towards ubiquitous computing environments, where sensor nodes tightly interact with actuators and behave based on the information surrounding them [4, 12, 13].

WSNs differ from conventional distributed systems in many aspects. Resource scarceness is the primary concern which should be carefully taken into account when designing software systems for WSNs. Sensor nodes are often equipped with a limited and nonrenewable energy source and a processing unit with a small memory capacity. Additionally, the network bandwidth is much lower than for wired communications and radio operations are relatively expensive compared to pure computation. The sensor nodes and network

are less reliable than in common network systems: depending upon the configuration of network and environment circumstances, wireless links may become degraded or unviable. These factors make the design of WSN applications very special and different from other networking technologies.

Among the major challenges in the development of WSNs, a primary one is *programming abstractions* for developing sensor software at different levels, ranging from drivers and operating systems, to network protocols, middleware services, and applications. Whereas the first two levels are characterized by close coupling between software code and hardware modules, the other upper layers are assumed to be programmed in some high-level programming models which may be different from those used for the first group. There have been several approaches to either address programming at a particular sensor software level, or provide a generic model that can be exploited to develop both low-level and high-level software functionalities [14].

A fundamental challenge in developing programming abstractions for WSNs is the careful consideration of inherent constraints in such platforms. These constraints include type of sensor applications (*e.g.*, real-time, mobile and database), scope of programming (node-level, group-level and network-level), and programming paradigms (*e.g.*, functional, imperative and event-driven). Indeed, the application type is the primary factor highly influencing the model of programming for WSNs. Beyond the detailed classifications presented in the literature for WSN application types, two general types have been envisaged, namely *static* applications and *dynamic* applications [15, 16]. In the static type, all application requirements are identified before developing and deploying the software on sensor nodes. In other words, the expected features from the sensor software never change during the lifespan of static applications, while in dynamic use cases WSNs deal with dynamic requirements and unpredictable future events and behave based on the context information surrounding them. Therefore, programming abstractions proposed for such environments should be enhanced with mechanisms that enable dynamic changes and modifications of the sensor software.

Principally, sensor software that is able to evolve during its life-time is referred to as *adaptive and reconfigurable* sensor software. Dynamic updating of sensor software can be applied to different levels of software abstractions, from operating system modules to application services. Bringing adaptivity to WSNs poses several major challenges to the development of applications in these types of networks. Programming models, code distribution, context information processing, adaptation reasoning, and reconfiguration mechanisms are the main challenges in this area. The existing resource limitations, as well as unreliability of wireless communications make addressing these issues quite difficult.

1.1 Problem Statement and Thesis Motivation

In many current WSN applications the nodes of the network are deployed in large number and inaccessible places for long periods of time. This introduces two main requirements in

terms of sensor software maintenance: *i)* in order to maintain long-lived WSN applications, we may need to remotely patch or upgrade software deployed on sensor nodes through the wireless network, and *ii)* a deployed WSN may encounter sporadic faults that were not observable prior to deployment, requiring a mechanism to detect failures and to remotely repair faulty code [17, 18]. The second part of the problem is related to the emerging WSN applications which deal with context-aware and ubiquitous environments. A sensor application, in such environments, should observe various types of ambient context elements, process them, and seamlessly adapt to the new conditions by reconfiguring the software functionalities [19, 20]. Additionally, in such heterogeneous applications, the sensor needs may require to communicate with different computing devices, thereby, the requirements from network configurations and protocols may change along the application lifespan [21]. Manually and physically updating of sensor software is not a feasible way to address the above concerns due to the scale and the embedded nature of the deployment environment, in particular when sensor nodes are difficult to reach physically. The problem statement of this thesis therefore is:

> *The existing limitations in WSNs such as sensor resources and network bandwidth make dynamic software updates on sensor nodes quite difficult to address. The main question is that how to enable remote update and reconfiguration of sensor software in an efficient way with respect to the WSN restrictions and various applications' requirements.*

The main goal of this thesis is to answer to the above question from WSN programming perspective in the context of SWISNET project. The state-of-the-art in this area is focused on low level reconfiguration issues and lacks a general framework supporting all essential aspects of application adaptation in sensor networks, including context-awareness, adaptation reasoning, and software reconfiguration. Whereas the first two issues have not been considered adequately, the third issue has drawn a considerable attention from the WSN researcher community [15, 16, 22, 21, 23, 24, 25, 26]. However, the early reconfiguration and reprogramming approaches were basically focused on reconfiguring the whole sensor software image [27] and the recent modular reconfiguration models [15, 16, 28] suffer from different drawbacks such as high resource usage and limitations in the range of use. To address the thesis question, we need to investigate the problem from different dimensions and explain how this thesis contributes to this research area of WSNs. A more precise consideration to the problem results in the following two questions:

1. *How to design an adaptation framework specialized for WSNs?*

   We define adaptation framework as changing the application behavior in a WSN system according to changes in surrounding context such as end-user needs, sensor resources and application logic. Therefore, the question is that what is the most efficient adaptation model for WSNs so that it can observe context information, reason on required adaptations, and implement the needed software reconfigurations. In addition

to adapting software behavior, the model should feature a resource-efficient technique to enable direct software updates for purposes such as fixing bugs and patching sensor software. As mentioned above, the existing approaches to WSN adaptation mostly focus on the reconfiguration issue and do not address the whole lifecycle of adaptive sensor applications. For instance, research on context information observation treats sensor nodes as context data collectors, rather than considering them as adaptable devices possessing their own contextual parameters [29, 30].

2. *How to abstract the software adaptation framework from other parts of the system (i.e., application and system software) in WSNs?*

   This question refers to the capability of proposing the software update mechanism as a well-described framework that can be plugged and integrated to the different types of software modules installed on sensor nodes, including low-level operating system modules and high-level application-specific services.

Among the challenges posed by these questions, programming models for WSNs and the associated software reconfiguration/reprogramming mechanisms are two fundamental issues that highly affect the efficiency of an adaptation framework for WSNs. These raise two new questions in this area, including:

3. *What is the appropriate sensor programming abstraction that meets the reprogramming requirements in WSNs?*

   Many programming paradigms have been proposed to develop static WSN applications. However, to support dynamic updates we need a modular programming model that provides the primitives required to identify the portion of software that needs to be updated, without affecting the other parts of the sensor software. Additionally, the generality of programming model—the possibility of using it across different hardware platforms and operating systems—is also an important issue as reconfiguration is essentially posed for ubiquitous sensor applications hosting different types of hardware and software products. The existing programming models that support dynamic modular updates suffer from different kinds of constraints such as high resource demands [28, 31] on sensor nodes, and lack of fine-grained reconfiguration [16].

4. *How to efficiently delimit and reprogram only the necessary portion of sensor software?*

   Ideally, a reprogramming model should be able to minimize the amount of rewriting in order to reduce the node-level and network level overheads. The latter refers to the energy required to transmit and receive a new update between sensor nodes, while the former concerns with the processing effort required to successfully reprogram a portion of sensor software and load the new code. Obviously, network-level overhead can be significantly mitigated by reducing the size of new code distributed across

the network as wireless data transmission is the dominant energy consumer in sensor platforms.

5. *How to distribute code updates across the network?*

   One of the primary requirements of any dynamically updatable WSNs is the mechanism by which the new code is distributed from a sink node, connected to a code repository machine, to other sensor nodes in a multi-hop network. The mechanism to select the sensor nodes targeted for updates, as well as the algorithm to propagate new code is a challenging issue.

Finally, the scope of our adaptation operation is not limited to only the sensor nodes in a homogeneous setting. Rather, it exceeds to the boundaries of variant computing devices in ubiquitous environments. Therefore, the last question of this thesis is:

6. *How to collect context information in heterogeneous WSN applications?*

   In heterogeneous settings, a mechanism to integrate sensor nodes with other heterogeneous entities becomes very essential. Specially, the context-processing part of the adaptation middleware needs this feature to be able to collect heterogeneous context information. Thus, to practically use the adaptation services, we need a distribution and integration middleware that provides a lightweight and yet general interaction model to connect sensor nodes to co-existing computing devices and network systems. Although there have been some valuable efforts to distribute sensor services and integrate them with common networks [32, 33, 34, 35], the state-of-the-art mostly focuses on low-level APIs and fails to provide a unified distribution abstraction relieving the programmer from dealing with the tedious and error-prone distribution tasks in WSNs.

1.2 Research Method

Generally, there are three major paradigms for the discipline of computing, including theory, abstraction, and design [36]. "modeling" and "experimentation" are two other substitutes for abstraction, however "abstraction" is more common for this paradigm in the discipline. Each of these paradigms is an iterative process consisting of four stages.

The first paradigm, theory, is rooted in mathematics. To development a coherent, valid theory the following four steps should be followed: *i)* definition: characterizing and defining objects of study, *ii)* theorem: hypothesizing possible relationships among obtained objects, *iii)* proof: determining whether the relationships are true; *iv)* interpreting results.

The second paradigm, abstraction, is rooted in the experimental scientific method and consists of four steps that are followed in the investigation of a phenomenon: *i)* forming a hypothesis, *ii)* constructing a model and making a prediction, *iii)* designing an experiment and collecting data, and *iv)* analyzing results.

The third paradigm, design, is rooted in engineering and consists of four steps that are followed in the construction of a system to tackle a given problem: *i)* identifying requirements, *ii)* stating specifications, *iii)* designing and implementing the system, and *iv)* testing the developed system.

Based on the identified research topics and goals, we adopt the design paradigm for the work presented in this thesis. We have first analyzed the requirements of the adaptation framework in the context of WSNs. To do so, we have carried out a systematic and exhaustive literature review on the dynamic aspects of WSN applications, then formulated the requirements of the system based on the main goals of the SWISNET project, the drawbacks of existing solutions, and future trends in the field of WSN adaptation.

At the second step, we have investigated the state-of-the-art in adaptation engineering and also adaptation management in other conventional network platforms. Then, we have proposed a set of specifications and criteria, such as system architecture, appropriate adaptation techniques for WSNs, programming models, and adaptation's core modules, to provide the prerequisites for developing the adaptation framework.

This step is dedicated to design the logical and physical models for each main part of system, including context management, adaptation framework, programming model, reprogramming model, and distribution framework. Afterwards, a real prototype (based on third-party HW components) has been built to demonstrate the developed solutions in the project. The results have also been compared to state-of-the-art solutions, and developed prototypes have integrated the acquired and accumulated knowledge.

Finally, we have tested and evaluated the prototypes in terms of technical feasibility, strengths, ease of use, resource efficiency, and network overheads, on our reference real hardware platform. Testing and evaluating on real sensor platforms is very important since the popular simulators and emulators are not mature enough to assess the programming proposals for WSNs and most of valuable and referable research efforts in this area are indeed developed based on a real prototype and even concrete application scenarios.

During the project period, research papers have been peer-reviewed by experts in the field. Presentations of the research results at international conferences and journals have also provided relevant feedback and opportunities for improving our proposals and exchanging ideas with other researchers. As part of our research result, the developed software modules have been made available as open source [37] in order to allow other researchers to repeat experiments and validate our results.

1.3 Results and Implications

The main contributions presented in this thesis have been published in a number of research papers [19, 38, 39, 40, 41] in relevant international workshops, conferences and journals. The concept that underpins our contributions throughout the thesis is the principles of Feedback Control Loop (FCL), as shown in Figure 1.1.

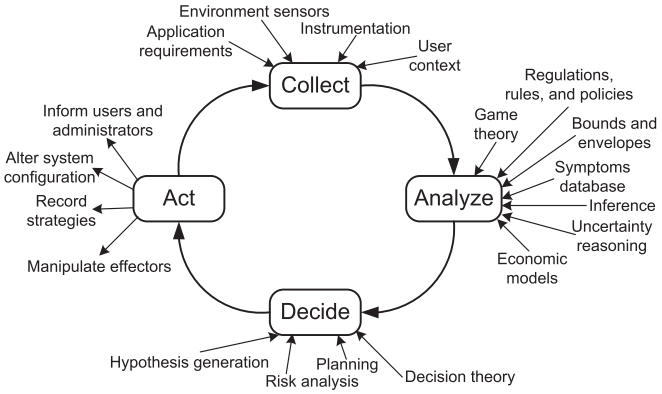

Figure 1.1: Activities of the feedback control loop.

FCLs have been recognized as important factors in software process management, software maintenance, and software evolution. A FCL typically involves four key activities: collect, analyze, decide and act [42]. The feedback cycle starts with the collection of relevant data from sensor nodes and other sources that reflect the current state of the system. Next, the system analyzes the collected data. Afterwards, a decision must be made about how to adapt the current system in order to reach a desirable state. Finally, to act against the decision, the sensor software must be reconfigured based on some reconfiguration mechanism.

The main results of this thesis are illustrated in Figure 1.2, where the software on a given sensor node is configured and communicate with other computing devices according to our adaptation and distribution solutions. The first set of results, represented as Adaptation Middleware, is aimed at addressing the basic adaptation requirements discussed in Question 1, Question 2 and Question 4 of the problem statement. In fact, these implement the main activities of FCL, including context monitoring and analyzing (Context Processor), adaptation reasoning (Adaptation Reasoner), and software reconfiguration (Reconfigurator). To implement this adaptation middleware, we need to answer Question 3 in the problem statement, referring to a programming model well-suited for adaptive WSN applications. This leads to the second result in Figure 1.2, where we propose a component-based Programming Model for static and dynamic WSN applications. Finally, the adaptation middleware is integrated to heterogeneous environments by the third set of results addressing a unified Distribution Model for WSNs.

It should be noted that the key results of this thesis along with extensive evaluation experiments are dedicated to the fundamental challenges—programming model, reconfiguration model, and distribution model. However, the results reported for context manage-

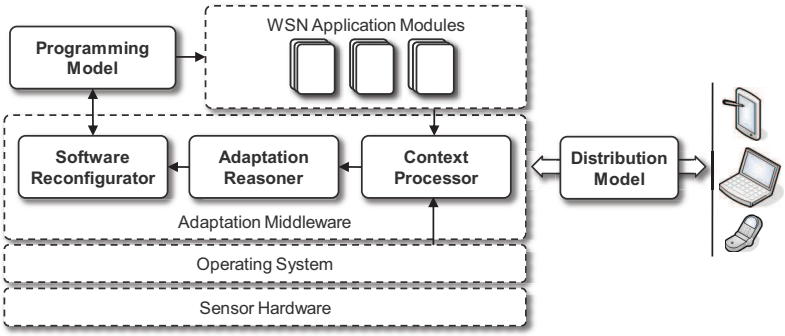

Figure 1.2: A general overview of the results of this thesis.

ment and adaptation reasoning are mainly focused on design models and the corresponding evaluation results are preliminary. It should be also stressed that the usefulness of results achieved by this thesis is not limited to a particular WSN application domain. On the contrary, we believe that the presented results are useful in most of WSN platforms requiring dynamic reconfiguration of sensor software.

In the rest of this section, we give a brief description of main contributions in this thesis, while the corresponding details are available in the mentioned published papers.

1.3.1 Reference Motivation Scenario

There are many different types of WSN applications that need to be dynamically adapted and maintained. As mentioned before, large-scale sensor applications as well as context-aware environments are in general subject to dynamic updates. This thesis focuses on *home monitoring systems* [43, 44]. The recent home surveillance systems are shifting from expensive cable-based infrastructures to easily deployable wireless systems, including WSNs. Sensor nodes in such environments can observe various types of ambient context elements such as temperature, smoke, occupancy, and also health conditions of inhabitant. Dynamicity in home monitoring scenarios is considered from two different perspectives. Firstly, such applications are characterized as long-lived applications, which may be deployed in a large number of dwellings by a vendor. To easily maintain (fixing bugs, patching security holes, and upgrading system and application software) the deployed sensor software in each home, the solution vender needs a central reconfiguration tools through which all monitored homes can be remotely maintained, instead of imposing the cost of physical maintenance to owners. Secondly, a wide range of sensors and actuators are needed in home systems to implement heating, ventilation, and air conditioning (HVAC) control. Since different sensor nodes are likely to run different application code and interact with

actuators, software reconfiguration may be needed to satisfy the dynamic requirements of owners.

1.3.2 Engineering Self-Adaptive WSNs through Feedback Control Loops (Chapter 7,8)

This thesis demonstrates that FCL is well suited for the domain of dynamic WSN applications, as argued in [19, 38]. To this end, we present a novel distributed middleware system, called WiSeKit, for addressing the dynamicity of WSN applications. WiSeKit's architecture reflects the main activities of FCL, including context-awareness (collect), adaptation reasoning (analyze and decide), and software reconfiguration (act).

To address the first and second activities, we propose a context management framework in the middleware layer of WSNs to process context information and provide the necessary analyzed data for adaptation and reconfiguration. This framework is inspired partially from the COSMOS framework—a comprehensive model for processing context information in ubiquitous computing environments [29]. In the proposed model, each piece of context information is defined as a context node. Context nodes can be considered as virtual sensor nodes that are distributed over the physical sensor nodes in the network with respect to the type of information provided by the context node and its role in the context model. The context processing framework consists of two main parts: *i)* a context information processing architecture for modeling the sensor context elements and their interactions, and *ii)* a middleware framework for executing the context model. To implement the context model, context nodes are mapped to the software components proposed specifically for context data processing in WSN applications.

The adaptation decision logic of WiSeKit is also inspired from the hierarchical architecture of typical WSNs. WiSeKit proposes three observation levels for adaptation reasoning: *local*, *intermediate* and *remote*. This is in accordance with the typical organization of nodes in sensor networks. In the lowest level nodes attempt to perform reasoning (local) on their own. At one higher level, cluster head (a more powerful node that transmits an aggregated sensor data to the distant base station) decides on an adaptation based on the data received from sensor nodes in its own cluster. Finally, the sink node, governing the whole sensor network, takes the action of reasoning concerned with the whole sensor network. At the last stage, WiSeKit addresses the software reconfiguration (act) through a component-oriented approach. It identifies two adaptation scopes to handle reconfiguration requests: parameter adaptation and component adaptation. Parameter adaptation supports fine tuning of applications through the modification of application variables and deployment parameters, while the latter allows the modification of service implementation (replacement of component), adding new components, and removing running components. WiSeKit is designed as a general adaptation solution for WSNs so that it can be used in a wide range of sensor applications, regardless of their domain. As mentioned before, these results reflect the overall design of our adaptation framework with preliminary evaluation,

while the rest of this section presents the key results.

1.3.3 Component-based Programming in WSNs (Chapter 9)

To meet the requirements of component-based reconfiguration within WiSeKit, we first propose a novel component-based programming for WSNs, then implement the reconfiguration goals over this component model. Our component model, called Remora, is a lightweight and event-driven component model designed for resource-constraint embedded systems, including WSNs. The philosophy behind Remora design is to: *i)* allow a wide range of embedded systems to exploit it at different software levels from operating system to application, and *ii)* to reify the concept of event as a first-class architectural element simplifying the development of event-oriented scenarios. The latter is one of the key features of Remora as a programming model for sensor networks that is expected to support exhaustively event-driven design. Reducing software development effort is the other objective of Remora. Remora components are described in XML as an extension of the Service Component Architecture (SCA) model [45] in order to make WSN applications compliant with the state-of-the-art componentization standards. It also features a C-like language to implement Remora components, attracting both embedded system programmers and PC-based developers to programming for WSNs. Remora has been successfully implemented and evaluated on the Contiki operating system [46].

1.3.4 Component-based Reprogramming in WSNs (Chapter 10)

To address component-based reconfiguration in WSNs, we reconsider the Remora component model in order to extend it with the capability of compositional component reconfiguration. The dynamicity of Remora is achieved by the principle of *in-situ reconfigurability*, which refers to minimizing the overhead of component reconfiguration through a set of in-situ updating guidelines. The run-time system that implements in-situ reconfiguration model is a middleware framework, called RemoWare. In particular, RemoWare supports the main features of our reconfiguration model, including new binary update preparation, wireless code propagation, run-time linking, dynamic memory allocation and loading, and component state retention. The middleware itself has been developed by the Remora component model on Contiki and the evaluation results show that RemoWare imposes a very low energy overhead in code distribution and component reconfiguration, and consumes approximately 6% of the total code memory on a TelosB sensor platform.

1.3.5 Sensor Service Distribution and Integration (Chapter 11,12)

The last contribution of this thesis is dedicated to providing a software framework for WSNs that enables programmers to develop distributed sensor services and integrate them with other conventional network systems at a high-level programming abstraction. In doing so, we leverage the concepts of component-based programming to design, describe and

implement distributed and interoperable WSN services. In particular, we reconsider the REMORA component model to extend it with the capability of distributing a component's services across the network. We also propose the integration of above component-based solution to a uniform interaction model in order to facilitate interoperability of sensor services with the Internet through Web service-enabled components. This interaction model is inspired by REST—an architectural style for distributed systems emphasizing scalability of component interactions, generality of interfaces, and independent deployment of components [47]. As mentioned before, the provision of such framework can be significantly useful in processing context information provided by heterogeneous nodes in the network.

1.4 Unaddressed Issues

Although this thesis attempts to address the main challenges identified earlier in this chapter, there are still some issues that are not considered or adequately addressed.

Concerning the questions posed from the problem statement, code distribution models (Question 5) are not investigated in detail as this issue has been extensively investigated by the WSN research community [27, 48, 49, 50]. However, prior to formulating a relevant code distribution model for a certain use-case, we need to establish a *code propagation substructure* that is designed specially for efficient distribution of code chunks to all or a subset of the sensor nodes. The thesis will contribute to this part of the problem.

Additionally, the implications of the research with respect to the privacy and security issues have not been considered. For example, in a military application, the sensor software may function improperly and insecurely when it is updated with unauthorized code sent by unknown users.

1.5 Structure of this Thesis

This thesis consists of an introductory part and six research papers. The role of the introductory part in a thesis based on a collection of papers is to provide a framework in which the work presented in the papers appears as parts of a whole.

The introductory part of this thesis is structured as follows. First, Chapter 2, 3, 4 and 5 describe related work within the four main areas addressed in this thesis, *i.e.*, self-adaptation in WSNs, programming paradigms for WSNs, sensor software reprogramming, and sensor service distribution. At the end of each of these four chapters we discuss some open issues which have been addressed by the work presented herein. Finally, in the concluding Chapter 6 we summarize the thesis, provide some critical remarks, and present some ideas and opportunities for further work.

Part II of the thesis contains the research papers. The papers appear according to the order of the results discussed in previous section, but depending on the familiarity with the different topics the reader may choose a different reading order. The reader is encouraged

to read the whole of Part I, which puts the papers into perspective.

Chapter 2

Self-Adaptation in Embedded and Ubiqouitous Systems

Growth in the usage of sensor nodes in ubiquitous computing environments, in the dynamism of the environments they operate in, and the need for timely adaptations as environmental conditions change, arise significant challenges for manual reconfiguration of sensor behavior. Therefore, there is increasing interest in adaptive WSNs that sense relevant contextual conditions and adapt automatically as they change. In this chapter, we first discuss the basic concepts of self-adaptive software systems and then study the state-of-the-art models proposed to incorporate adaptation to embedded and sensor systems in ubiquitous settings.

2.1 Background

The complexity of existing software development technologies and uncertainty in software systems have led the software engineering community to consider inspiration in diverse related fields, such as artificial intelligence, control theory, and robotics, in order to address design and management of software systems [51]. One of the most promising research topics in this area has been the capability to accommodate software system's behaviors in the form of self-adaptation—systems that are able to adjust their behavior in response to their perception of the environment and the system itself. The "self" prefix indicates that software systems decide autonomously how to adapt in their contexts and environments, without or with minimal human interference. While some self-adaptive systems may be able to run without any human intervention, feeding adaptation frameworks with information about higher-level objectives is useful in many dynamic systems.

The concept of self-adaptivity applies to the research in several application areas and technologies such as autonomic computing, dependable computing, embedded systems, mobile networks, multi-agent systems, peer-to-peer applications, sensor networks, service-

oriented architectures, and ubiquitous computing. The proper realization of the self-adaptation functionality is a significant challenge and it is very important to take into the consideration all characteristics and limitations of the target computing platform when studying self-adaptation.

The emergence of highly distributed embedded systems, which are often long-lived, has made it highly infeasible to manually manage and control software systems installed on such systems. For instance, embedded systems play an important role in many mission-critical applications such as NASA Apollo mission. Many embedded systems either lack the interfaces to enable software upgrade, or their missions inherently require autonomous adaptivity as the latency induced by the communications makes human intervention not applicable [52]. There are growing interests in self-adaptive software for embedded application areas such as automotive systems, smart phones, and sensor and actuator networks. Despite of its benefits, developing self-adaptive embedded software turns out to be quite difficult. Self-adaption incurs more dimensions of complexity to system design such as dependability and fault-tolerance, which makes it much harder to implement and validate a self-adaptive embedded software. Lack of a formal process also contributes to difficulty in developing high quality self-adaptive embedded systems.

In a higher level of abstraction, ubiquitous computing, encompassing embedded systems, is an exciting paradigm shift towards information technology that is invisibly woven into our surroundings. It promises ubiquitous access to information and services seamlessly integrated into our daily life activities. This arises the need for context-aware, adaptive applications that can be adjusted at run-time in order to make use of the changing execution context. In addition to conventional context dimensions such as resource availability, physical location, network connectivity, and battery status, in ubiquitous computing environments services may appear and disappear while the user is moving. Thus, ubiquitous applications are assumed to dynamically adapt their behavior in order to improve their functionality and quality of service.

2.2 Basic Concepts

This section presents a general review of the basic concepts and principles in self-adaptive software. It also discusses a classification of existing adaptation mechanisms and policies.

2.2.1 Self-* Properties

Adaptivity properties are often known as self-* properties in the literature. One of the most well-known set of self-* properties, identified by IBM, include eight properties [53]. This section discusses these properties, along with some other related issues, which have been considered by the research community in this area.

The hierarchy of adaptation self-* properties can be organized in three levels, including general, major, and primitive levels.

General Level. This level contains global properties of self-adaptive software, including self-managing, self-governing, self-maintenance, self-control and self-evaluating.
Major Level. A set of four properties is defined at this level. These properties have been specified in accordance to biological self-adaptation mechanisms, *e.g.*, the human body adapts itself to changes in its surrounding context (*e.g.*, changing temperature in the environment) with the similar properties. The main properties in this category include:

- *Self-configuring* refers to the capability of reconfiguring automatically and dynamically in response to changes by installing, updating, unloading, integrating, and composing/decomposing software entities.
- *Self-healing* indicates the capability of discovering, diagnosing, and reacting to disruptions. This includes two detailed properties: self-diagnosing and self-repairing. The former refers to diagnosing errors, faults and failures, while self-repairing focuses on recovery from them. Self-healing can also anticipate potential problems, and accordingly take proper actions to prevent a failure.
- *Self-optimizing*, which is equivalent to self-tuning or Self-adjusting, is the capability of managing performance and resource allocation in order to satisfy the performance requirements of end-user. Important concerns related to this property are end-to-end response time, throughput, utilization, and workload.
- *Self-protecting* is the capability of detecting security threats and recovering from their effects. It has two aspects, namely defending the system against malicious attacks, and anticipating security-related problems and taking actions to avoid them or to reduce their effects.

Primitive Level. The primitive properties are self-awareness, self-situated, and context-awareness. *Self-Awareness* is used for systems that are aware of their self states and behaviors, while a *self-situated* system must be aware of its current external operating conditions. *Context-Awareness* means that the system is aware of the current contextual situation and changing circumstances.

2.2.2 Requirements

To make a software system adaptive, there are six questions that should be answered [54]. These questions are considered during the requirement analysis phase of engineering adaptive software.
Where: This question is concerned with where changes are needed. In particular, the question is that which artifacts of system at which software layer (*e.g.*, system or middleware) and level of granularity need to be changed. To answer the question, we need to investigate attributes of adaptable software, its software architecture, and coupling between its modules and layers.

When: Temporal aspects of adaptation are addressed by this set of questions. When does a change need to be applied, and when it is feasible to perform that? Can a change be made once the system requires, or are there constraints that limit temporal changes? How often does the system need to adapt? Is there any temporal rule for the changes happening continuously, or do they occur only as needed? Is it enough to perform adaptation actions reactively, or do we need to predict some changes and act proactively?
What: Answers to this set of questions identify what attributes or portions of the software system are subject to change. There is a wide range in terms of change granularity, from parameters and methods to components, architecture style, and system resources. It may be also important to identify the range of changes (*e.g.*, parameter values). It is also important to specify what artifacts and attributes should be monitored to follow-up on the changes, and what resources are essential for adaptation actions? Regarding the distinction between the what and where, Where identifies which part of the system caused the problem, while what specifies the attributes and artifacts that need to be changed to resolve the problem, *e.g.*, in a WSN application, it is essential to know which part caused energy performance degradation (*e.g.*, the radio due to transmitting too much data) and after that, what needs to be changed (*e.g.*, changing the sensing sample rate).
Why: This question refers to what is the main objectives of carrying out adaptation, *e.g.*, robustness or resource-efficiency.
Who: This question identifies the level of automation and human intervention in self-adaptive software. Although in a self-adaptive system it is expected that there will be minimum human involvement, having an effective interaction with end-users may be required to inject adaptation policies to the system and therefore improve the efficiency of adaptation decision system.
How: The last but not the least, the mechanisms, and strategies by which the adaptable artifacts can be changed is of vital importance. There are many technical issues that should be considered in answering this question based on the architecture of adaptive system, its characteristics and limitations.

Prior to developing self-adaptive software, the developer needs to find proper answers for the above questions in order to identify mechanisms and alternatives used in implementation. Some of these questions may be answered by end-users and managers through adaptation policies, and the rest should be determined by the adaptive system itself.

2.2.3 Adaptation Policies

One of the intrinsic problems in self-adaptivity is the selection, the calculation or the derivation of the new configuration that fits the current state of the system actors. Depending on the system and the targeted application, this process may be realized in different forms by adopting different adaptation policies. We distinguish between three kinds of policies, namely action-based, goal-based, and utility function based policies.
Situation-action Adaptation. Action-based policies [55] are the most popular form

and are used in different domains related to networks and distributed systems such as computer networks, active databases and expert systems. An action policy consists of situation-action rules which specify exactly what to do in certain situations.

Goal-oriented Adaptation. Goal-based adaptations [56] are a higher-level form of behavioral specification that specifies performance objectives, leaving the system to take the actions required to achieve those objectives. This is similar to dynamic systems in which mechanisms are needed to allocate and control computational resources to guarantee promised levels of QoS. Since goals provide only a binary classification into *desirable* and *undesirable* performance, goal-based adaptation models mostly focus on maximizing the probability of achieving goals or minimizing the degree to which goals are not met.

Utility Function-based Adaptation. This policy description model is an extension of goal-oriented approaches. Utility functions [57] describe a real-value scalar desirability to system states. Therefore, utility functions express the rationale for adaptation decisions in a precise way, and are therefore more accurate than goal policies when adaptation triggers, or when goals are in conflict. Although utility-based approaches provide precise decision making, they force developers to specify in detail the properties of the application variants. The utility function computes the utility of an application variant based on the given properties of the different variants and the current context. The result is the application variant that maximizes the utility of the application while satisfying the resource constraints provided by the underlying environment.

2.3 Existing Adaptation Solutions

Self-adaptation of embedded and ubiquitous software systems is still a young research area. The main hurdle in addressing adaptivity in embedded systems is perhaps the high resource demand of traditional adaptation solutions. In fact, the complexity of typical adaptation models needs a careful reconsideration to tailor them for resource-poor embedded platforms. In this section, we present research works making significant contribution to this area, while most of the proposed approaches have been devoted to mobile platforms as contextual changes in mobile applications are quite frequent and unexpected.

2.3.1 RoSES

The Robust Self-Configuring Embedded Systems (RoSES) project [58] is an early endeavor to bring adaptivity to embedded systems. The main goal of the RoSES project is to create robust, flexible, maintainable, distributed embedded systems that support graceful degradation via in-service software reconfiguration. Using RoSES framework, an embedded system would be able to automatically reconfigure to accommodate failed, upgraded, or inexact spare components. The RoSES project envisions a system of “smart” sensors and actuators connected to an embedded real-time network, where every sensor acts as a “server” to any node desiring its functionality, as shown in Figure 2.1. The core of

the framework in a sensor node is a software adapter to translate between architected state variables on the system network and the sensing capabilities of various sensors and actuators. A customization manager is also designed to maximize system-level functionality by addressing an optimization problem of allocating a subset of possible functionality to maximize overall utility.

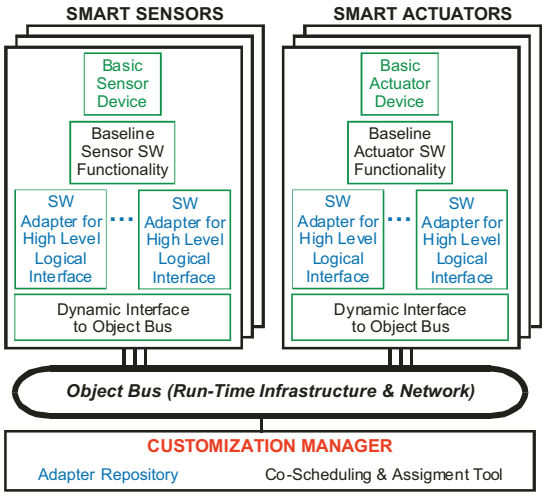

Figure 2.1: The generic architecture of RoSES framework.

However, the RoSES project fails to consider the modeling of dynamic adaptation. Rather, it addresses dynamic adaptation by leaving all configuration decisions to the system. Therefore, the developer does not need to care about the reconfiguration process, but the adaptation behavior can hardly be analyzed. Additionally, dynamic adaptation model in the RoSES framework is essentially restricted to the software architectural level.

2.3.2 Robocop, Space4U and Trust4All

The goal of these projects, which followed up each other, is to propose a component-based software architecture for the middleware layer of high volume embedded appliances. The building blocks of the middleware software is the Robocop component model inspired by COM [59], CORBA [60] and Koala [61]. A Robocop component is a set of possibly related models. Each model provides a particular type of information about the component.

The aim of the Robocop project [62], in particular, is to develop a middleware architecture and component model for high volume consumer electronics, *e.g.* mobile devices and PDAs. The proposed architecture supports robustness, real-time applications, secure

component downloading and upgrading. Space4U [63] aims at consolidating and extending on the results of the Robocop project by investigating a number of research issues that are considered extremely important for the embedded application domain, but have not been covered by Robocop. The Space4U project extends the middleware architecture proposed in the Robocop project with fault prevention, power management, and terminal management. Finally, the main goal of the Trust4All project [64] is how to establish and maintain correct operation of a system, while the software embedded in the system is able to be upgraded and extended dynamically while the system is in use by a customer. All together, these projects provide a core architecture comprising adaptive mechanisms for the management of the above characteristics and support the Robocop component model.

2.3.3 CARISMA

CARISMA [65] is a mobile computing peer-to-peer middleware solution leveraging the principle of reflection to support the construction of context-aware adaptive applications. Specifically, CARSIMA offers applications a middleware framework as a dynamically customizable service provider. Customization takes place through metadata, which change middleware behavior to answer application service requests in various contexts and trigger the adaptation of the deployed applications by detecting execution context changes. CARISMA uses utility functions to select application profiles, which is used to select the appropriate action for a particular context event, with the risk, however, of incurring conflicts. To tackle conflicts, CARISMA features a micro-economic approach that relies on a particular type of sealed-bid auction. It treats a distributed mobile system as an "economy" where applications compete to receive the quality-of-service they desire. In this economy, the middleware plays the role of an auctioneer, collecting bids from applications and selecting the policy that maximizes social welfare. This approach is particularly appropriate for the mobile setting as it meets the requirements of dynamicity, simplicity and customizability that are typical of this environment.

2.3.4 MADAM

The Mobility and Adaptation-enabling Middleware (MADAM) project [66] aims at providing software developers with reusable models and tools, assisting them in the design and implementation of adaptive, mobile applications. The MADAM project follows an architecture centric approach where dynamic adaptation is realized in an application independent middleware. The core idea in MADAM is to transfer software product line concepts, usually used for design-time variability management, into run-time. The result is a run-time representation of the applications' architectural models which serves as a basis for an adaptation manager component to reason about and to control adaptation. To enable the reusability of adaptation strategies, MADAM proposes a middleware layer which can be used to encapsulate context monitoring, adaptation reasoning logic and reconfiguration tasks.

Figure 2.2 shows the main ideas of the MADAM adaptation approach. The adaptation middleware realizes three main adaptation functions: context management, adaptation management, and configuration management. To accomplish these functions, the adaptation middleware requires knowledge about the application structure and constraints, as well as the various context and resource dependencies. This means that the middleware can be used in dynamic applications in which all adaptation variants and constraints can be specified at run-time through the application architecture model. Applications' modules are also modeled as component frameworks where functionality defined by a component framework can be dynamically configured with conforming component implementations.

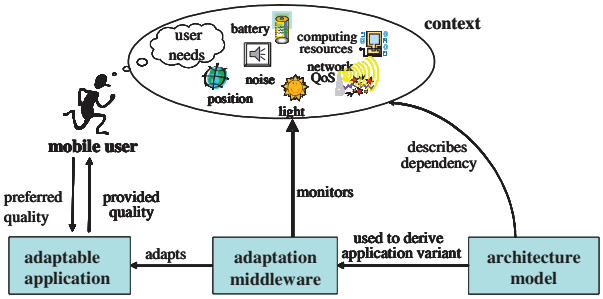

Figure 2.2: MADAM's approach to adaptation in mobile environments.

2.3.5 MUSIC

The MUSIC planning framework introduced is an extension of the MADAM planning framework, which supports the adaptation of component-based architectures [67]. The main goal of MUSIC is to maintain Quality of Service (QoS) in mobile environments through an application independent middleware approach. MUSIC aims at separating the self-adaptation concern from the business logic concern and therefore delegates the added complexity related to self-adaptation to generic middleware. The adaptation process, in MUSIC, relies on the architecture model of the application, which specifies its adaptation capabilities and its dependencies to context available at run-time. Similar to MADAM, the application development model is a component framework, which defines the functionalities that can be dynamically configured with conforming component implementations. MUSIC features an adaptation-planning framework to evaluate the utility of alternative configurations in response to context changes, to select a feasible one for the current context and to adapt the application accordingly. Thanks to its enhancements in supporting service-oriented architecture, the planning middleware is capable of exploiting remote services as well as local components and services to maximize the overall utility of the applications.

Figure 2.3 depicts the component-based architecture of the MUSIC platform. Adaptation manager as the heart of MUSIC consists of Adaptation Reasoner (to support the execution of the planning heuristics, which is driven by metadata included in the plans), Adaptation Controller (to coordinate the adaptation process) and Configuration Executor (to reconfigure the application based on the set of plans selected by the planner).

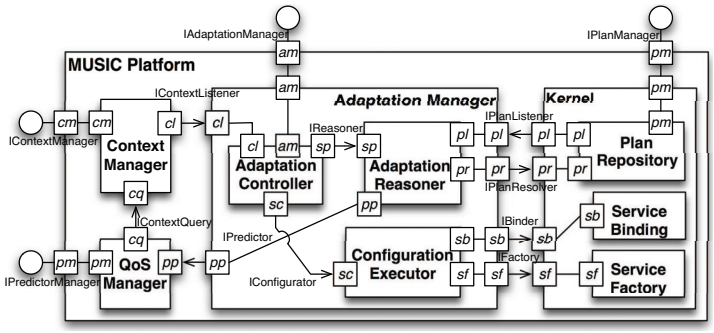

Figure 2.3: Architecture of the MUSIC adaptation platform.

2.3.6 Agilla

Agilla [68] is a mobile agent middleware designed to support self-adaptive applications in sensor networks. Agilla builds up applications out of mobile agents, which are special processes that can explicitly migrate or clone from node to node while carrying their state. Agilla is particularly suited to handle situations in which local decisions would significantly reduce the amount of data wirelessly transmitted. For example, in a fire tracking application, to prevent having to coordinate the application from a sink node, mobile agents can be employed to autonomously and locally adapt to the changing location of the fire. Therefore, it relieves the application installer from needing to deploy a service on every node in the network.

The overall adaptation solution of Agilla is presented in Figure 2.4. Each node supports multiple autonomous mobile agents and interactions among them are provided by two basic data abstractions on each node: a neighbor list and a tuple space. A tuple space is a type of shared memory in which data is structured as tuples accessible via pattern-matching. Proposing tuple spaces in Agilla enables agents to function autonomously and migrate while still being able to communicate. Tuple spaces can be accessed by agents residing on the same node, as well as to agents residing on different nodes.

Agilla proposes two forms of migration, *strong* and *weak*, to support the variant self-adaptation needs of WSN applications. Strong migration is useful when performing computations that span multiple nodes. It transfers both the code and state, allowing the agent

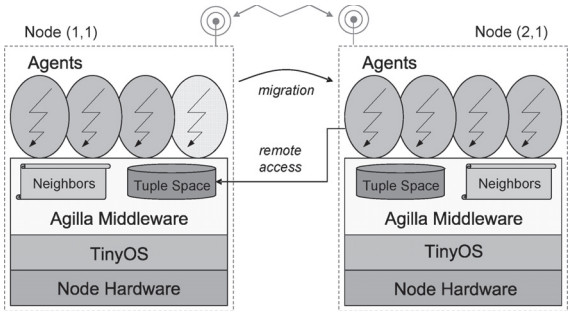

Figure 2.4: The Agilla model to address self-adaptation in WSNs.

to resume execution at the destination. Contrary to strong form, weak migration only migrates the code, imposing less overhead since the state does not need to be transferred, but implies to reset the agent.

2.4 Discussion

Our focus in this chapter has been on the efforts made to enable self-adaptation in embedded, mobile and ubiquitous systems. WSNs as a resource-constrained member of ubiquitous systems pose their own challenges when the software deployed on nodes are subject to changes. In other words, to systematically address the adaptation concerns in WSNs, we need to revisit the principles of FCL from a different perspective and study how they should be properly described and tackled. Unfortunately, the state-of-the-art has hitherto fallen short of providing a comprehensive adaptation model for sensornets.

The first shortcoming occurs in the first activity of FCL, where we need a context management framework to not only monitor the contextual parameters of the environment, but also observe the proprietary context elements of sensor nodes. Most of frameworks discussed in this chapter consider sensor nodes as a context collector entity. In WSNs, the context management framework should be able to model different types of context data, ranging from sensor resources to environmental changes. To address adaptation policy, we can rely on the de facto solutions available in the literature such as situation-action. Basically, adopting the appropriate adaptation policy is largely dependent on the nature of application, even though we believe the majority of WSN applications can follow the simple situation-action fashion. However, the critical issue in this context is consideration of WSN architecture. In particular, the distributed and hierarchical organization of sensor nodes in typical sensor applications should be utilized to make more accurate adaptation decisions. These issues have motivated us to propose a generic adaptation solution for WSNs that takes into account their special characteristics when addressing FCL activities.

Chapter 3

Programming Wireless Sensor Networks

Ease of programming has long been recognized as a major hurdle to the wide-spread adoption of WSNs. WSN programming has been quite difficult due to not only its inherently distributed nature, but also the presence of severe operating conditions such as highly constrained resources, unreliable network communications, and faulty nodes. In this chapter we, therefore, present some background information regarding programming in sensor networks. First, we briefly discuss principles of WSN programming, as well as taxonomy of programming abstractions for sensor platforms. Then, the existing high-level programming approaches for sensor systems are reviewed.

3.1 Background

Most of early WSN applications were developed in an ad-hoc manner such that the sensor software was designed in a rather unstructured way and the application logic was intertwined to the underlying system interfaces. This forces programmers to deal with low-level system libraries, as well as with programming low-level networking interfaces. Moreover, developers' expertise in state-of-the-art programming models becomes useless when sensor nodes are programmed without any well-established discipline of program specification. This has drawn much attention in the WSN research community and motivated researchers to study sensor programming models as a hot topic in this area.

In this endeavor, a significant number of programming approaches have been proposed for the development of applications with different characteristics and requirements. Therefore, to choose an appropriate programming model and platform for a particular application, we need to investigate the application requirements, as well as the basic differences between programming abstractions. Generally, the approaches to programming sensor networks can be classified into low-level programming models and high-level programming

models. The former is focused on abstracting hardware and providing a set of basic libraries required to develop applications. For example, programming models for sensor operating systems fall in this category, *e.g.*, NesC language for TinyOS operating system [69]. Such programming models are often promoted to be utilized for developing application modules as well. High-level models address the programming issues from application point of view instead of low-level system concerns. They essentially focus on simplifying the development of application logics, relieving the programmer from the burden of dealing with low-level issues. High-level programming models are either an extension of a particular low-level programming model, or completely different from low-level languages with a new set of abstractions, as well as with mechanisms to integrate to system software.

3.2 Requirements

In addition to the constriants identified by a particular application, there are basic requirements for programming in sensor networks which are common for most WSN application domains. In this section, we highlight the main requirements of WSNs from programming point of view.

Similar to all other research challenges in WSNs, *resource-efficiency* is the primary concern when proposing a new programming model. By resource, we mean the memory requirements (code memory and data memory) of a programming model, as well as the energy required to run an application written by a particular programming abstraction. The latter makes sense when programming models aim to address network-level issues, such as distributed communication and group-level programming. The former, instead, focuses on node-level memory resources consumed by the constructs and run-time system of programming models. As such types of resources are extremely limited, the structure of programming models are often sacrificed to preserve the resource-efficiency. Thus, addressing this issue becomes very important when designing a high-level development model for sensor networks.

The other concern is related to the scalability of sensor networks, where an application may run over hundreds or thousands of sensor nodes. Due to bandwidth constraints, the amount of communications among nodes should be reduced in order to use efficiently the limited bandwidth and therefore improve the scalability of network. Programming models play an important role in this problem. They should provide programmers with scalable programming constructs to avoid transmitting large size program data across the network.

Supporting asynchronous and event-driven programming is also a critical requirement for WSN applications and embedded applications which are inherently event-driven. Sensor nodes are essentially reactive systems which are mainly driven by events. Specifically, their progress as a software system depends on external and internal events, which may occur unpredictably and unexpectedly at any time and at any rate. Sensor nodes need to be able to sense events in the physical environment and to react to them appropriately. They also require reacting to network messages from other nodes due to the tight collab-

oration of nodes within WSNs. Finally, sensor nodes should observe a variety of internal events, such as interrupts generated by sensors, or low battery indications. Therefore, sensor programming models should feature a robust and reliable technique to enable the programmer to easily implement the event-related logic of the system. In some programming approaches, events are considered as high-level programming constructs, providing a simpler way to control and process application events. In contrary, many other models do not specifically propose an event-driven approach of programming, but events are hidden from the programmer's view and handled implicitly by the programming model's run-time system.

Sensor software is error-prone and repairing the faulty nodes is a cumbersome task. Beside the fault recovery mechanisms devised in operating system and hardware, the programming model should also provide the primitives to avoid generating errors and handle the unexpected faults during application run-time. Making application failure-resistant needs support from programming models as it is very difficult and time-consuming for the programmer to take care of all error-prone code and handle failures that are unexpected and unknown before the deployment.

3.3 Taxonomy of Programming Models

In this section, we briefly discuss the prominent programming paradigms for sensor networks, categorized into *agent-oriented*, *component-based*, *event-driven*, *imperative*, and *functional*. We organize these models based on a few surveys reported in the literature on classifying WSN programming models [14, 70], as well as the main concerns of this thesis. Specifically, the main focus on this section is on structure of programming models for WSNs, rather than considering the high-level programming techniques proposed for a particular concern, such as network, storage-centric, and group-based programming.

3.3.1 Agent-oriented

The term Agent-oriented Programming (AOP) was coined in 1989 by Yoav Shoham [71] to describe a new programming paradigm. An agent oriented view of a software system consists of an implementation problem requiring multiple agents to represent the decentralized nature of the problem. In the agent-oriented programming model, agents as autonomous problem-solving entities interact with other agents through a high-level communication model. Whereas objects in object-oriented programming communicate through simple method invocations or method calls with limited range of variation in parameters, agents communicate through a declarative high-level agent communication language. From the network point of view, agents can migrate from one node to another node in a network (*i.e.*, mobile agents). To preserve program consistency, the state of the running program is saved, transported to the new node, and restored, allowing the program to continue where it left off.

Agilla. Agilla is a middleware for WSNs that provides a mobile-agent style of programming [68]. Agilla applications are composed of one or more software agents that can proactively migrate their code and state across the network. Agilla programming model provides two forms of migration, strong and weak. The former transfers both the code and state (for computations spanning multiple nodes), allowing the agent to resume execution at the destination, while weak migration only migrates the code. It is useful for performing computations that span multiple nodes. Local coordination among agents is accomplished using tuple spaces—a type of shared memory in which data is structured as tuples that are accessed via pattern-matching. This enables loosely-coupled communications between sensor nodes since sender and receiver do not need to agree on a shared memory address. Agilla therefore provides a powerful mechanism to implement applications requiring adaptation of some functionality in response to external phenomena, as mentioned in the previous chapter. As a motivation application, in a fire monitoring application, when temperature value exceeds the given threshold, fire-detection agents spawn fire-tracking agents that swarm around to collect information about the exact location of the fire.

3.3.2 Component-based Programming

Component-Based Software Engineering (CBSE) [72] has been recognized as a well-structured programming model to develop software systems. Component-based programming provides a high-level programming abstraction by enforcing interface-based interactions between system modules and therefore avoiding any hidden interaction via direct function call, variable access, or inheritance relationships. This abstraction rather offers the capability of black-box integration of modules in order to simplify configuration and maintenance of software systems. Module reusability and provision of standard API are some other advantages of adopting component-based software development [72, 73]. There have been a few significant efforts to enable CBSE in WSN applications. Whereas a very few of them have been successful and used for developing many real applications, some other approaches have not been promising due to reasons such as inducing high resource usage on typical sensor nodes.

nesC. nesC is perhaps the most popular programming model in this category and certainly the most successful one [69]. nesC is an event-driven programming language for WSNs derived from the C language. It was originally proposed to develop the TinyOS operating system—the most popular system software for sensor nodes. nesC is a structured component-based dialect of C designed for building embedded systems, specially for sensor networks applications. An application in nesC consists of a set of components linked together with bidirectional interfaces that serves as access point to the component. This makes program structure considerably more readable and helps to enforce good programming practices. Two types of components can be defined in nesC, namely configuration and module. A configuration indentifies the assembly of application components, while modules represent the component implementation. A nesC-based application is described

by a top-level configuration that wires components. The nesC language also defines a concurrency model based on tasks and hardware event handlers. In nesC, code can be run one of two ways: asynchronously from an interrupt handler, or synchronously as part of a scheduled 'task'. The creators of nesC are particularly concerned about 'data races' resulting from concurrent attempts to access a single variable. If it is necessary to ensure that particular block of code cannot be interrupted, we can declare a section of code "atomic."

OpenCom. OpenCom [28] is a generic component-based programming model for building system applications without dependency on any target-specific platform environment. This is achieved by splitting the programming model into a simple, minimal kernel, and then providing on top of this a principled set of extension mechanisms that allow the necessary tailoring. OpenCom components interact with other components exclusively through "interfaces" (provided behavior) and "receptacles" (interfaces that make explicit the dependencies of a component on other components). Connections between interfaces and receptacles are established through *bindings*. Since OpenCOM is constructed upon a minimal microkernel, it has the capability to be used as a programming model for WSN applications. Moreover, the C implementation of OpenCom is designed as a minimal implementation for resource-constrained devices. As a real application, Gridkit [74] is an OpenCom-based framework for sensor networks, realizing co-ordinated distributed reconfigurations based on policies and context information provided by a context engine. This system was deployed on Gumstix-based [75] sensor platforms (a resource-rich node type) for a flood-monitoring scenario.

Think. Think is an implementation of the Fractal component model [76] that aims to consider the specific constraints of embedded systems, including WSNs. The Fractal specifications define a hierarchical, reflective and general-purpose component model. A component definition exports functional interfaces (provided or required), configuration attributes, and may also provide non-functional interfaces implementing introspection and architectural reconfiguration services at run-time. The Think framework [31, 77] allows developers to build embedded systems and WSN applications made out of Fractal components. A system architecture is described using an Architecture Description Language (ADL), interfaces are defined using an Interface Description Language (IDL). The code that implements the method of server interfaces is written in regular C where ADL symbols are represented by convenient C symbols. The choice of the Think framework is motivated by the fact that it allows fine-grained reconfiguration of components.

LooCI. LooCI [78] introduces a loosely-coupled component infrastructure, which features an event-based binding model inspired by event-driven programming models, Service-Oriented Architectures (SOA), publish/subscribe interaction models and pluggable networking support. LooCI offers support for two component types, macrocomponents and microcomponents. The former refers to coarse-grained and service-like, building upon the notion of Isolates inherent in the SQUAWK [79] virtual machines. Isolates are process-like units of encapsulation and provide varying levels of control (depending on the specific JVM) over their execution. Each macrocomponent runs in a separate isolate and com-

municates with the run-time middleware via Inter Isolate RPC (IIRPC), which is offered by the underlying VM. Microcomponents are instead fine-grained and self-contained unit of functionality. All microcomponents run in the master Isolate alongside the LooCI runtime. LooCI component model also offers run-time reconfiguration, interface definitions, introspection and support for the re-wiring of bindings.

3.3.3 Event-driven Programming

The event-driven programming model can be considered as a conceptualization of the control-loop approach. This programming model can also be described as an application architecture technique in which the application has a main loop which is clearly divided down to two sections: the first is event detection, and the second is event handling. In particular, it enforces the separation of event detection and event handling by making the control loop an integral part of its run-time environment. The event-driven model is the most popular programming approach for WSN applications today. Most of programming models we discussed above are essentially event-driven, such as TinyOS and nesC, LooCI, Agilla, and Regiment. In this section we investigate other existing event-driven programming models for WSNs.

Contiki-ProtoThreads. Protothreads [80] are originally conditional blocks inside C functions which represent lightweight stackless threads. They provide a blocking context on top of an event-driven system, without the overhead of per-thread stacks. The main objective of Protothreads is to implement sequential flow of control without using complex state machines. In fact, Protothread is designed to reduce the memory cost of thread in traditional multi-threading system in which a thread requires its own stack with high memory requirements. A Protothread runs within a single C function and cannot span over other functions. It may call normal C functions, but cannot block inside a called function. In principle, Protothreads are similar to asymmetric co-routines. However, each co-routine has its own separate stack, while Protothreads are stackless.

DS-Ware. DS-Ware [81] is a message passing programming model whose focus is the real-time detection of sporadic events. It uses a form of publish/subscribe in which users specify subscriptions based on the attributes of the phenomena of interest. A higher-level of event handling is provided that programmers can use to infer the occurrence of specific events from the raw sensor observations, *e.g.*, an event can be defined as the composition of two physical sub-events occurring within specific time interval one from the other. Subscriptions are expressed using a dialect of SQL from the user machine.

3.3.4 Imperative

Imperative programming paradigm describes computation in terms of a program state and statements that change the program state. In imperative programming, a variable may be assigned to a value and later reassigned to another value. The collection of variables and the associated values and the location of control in the program constitute the state. The

state is a logical representation of storage which is an association between memory locations and values. A program in execution generates a sequence of states and the transition from one state to the next is determined by assignment operations and sequencing commands. Using imperative programming model, the programmer writes code that describes the exact steps the computer must take to accomplish the goal which is sometimes referred to as algorithmic programming. This paradigm is used in opposition to declarative programming, which expresses what the program should accomplish without specifying how to do it in terms of sequences of actions to be taken.

Abstract Task Graph. The Abstract Task Graph (ATaG) [82] is a mixed imperative-declarative programming model for application development on sensor networks. In particular, ATaG builds upon the core data driven concepts and incorporates novel extensions for distributed sense-and-respond applications. The application functionalities in the system are modeled as a set of abstract tasks with well-defined input/output interfaces. In addition, a set of abstract data items are defined to represent types of information exchanged between abstract tasks. Input and output relationships between abstract tasks and data items are explicitly described as channels. Each abstract task is associated with code developed by the user, implementing the actual information processing functions in the system. An ATaG program is called 'abstract' because the number and final placement of tasks and the coordination mechanisms are determined at compile-time and/or run-time depending on the characteristics of the target application.

An ATaG program represents an architecture-independent specification of the application functionality. In this way, the developer has the ability to specify application behavior for a generic, parameterized network architecture, thereby, the same application may be automatically assembled for different network deployments, or adapted as nodes fail or are added to the system. This also allows development of the application to proceed at the edge of deployment prior to decisions being made about the final configuration of the nodes and network.

Figure 3.1 shows an ATaG program for an environment monitoring application. The application is deployed on a network of sensor nodes, each equipped with a temperature and a pressure sensor. The application is designed to periodically compute and log the maximum pressure in the system, as well as to periodically monitor the environment temperature. If the temperature gradient between a node and its neighbors exceeds a threshold, the node is required to investigate the anomaly by surveying a larger area and then trigger an alarm.

Pleiades. Pleiades [83] enables a centralized programming approach in sensor network so that the central application has access to the entire network. Pleiades employs a program analysis for partitioning central programs into node-level programs and for migrating program control flow across the nodes. Pleiades also provides a simple construct, called cfor, to allow a programmer to introduce concurrent executions at multiple nodes. Whenever required, the Pleiades run-time system guarantees serializable execution of cfor statements.

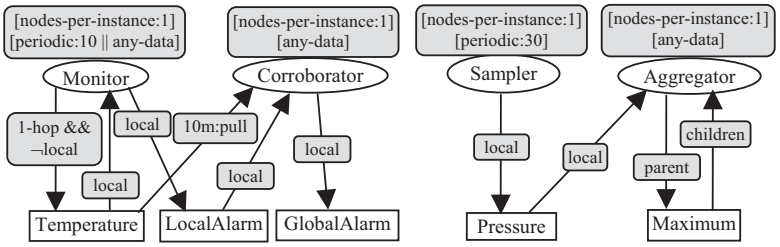

Figure 3.1: An ATaG program for environment monitoring.

3.3.5 Functional Programming

Functional programming is a model of programming in which the primary method of computation is the application composed of functions. In particular, functional programming involves creating and manipulating functions to build up larger programs. This requires a language that allows functions to be used as input and return data to other functions. One of the motivations for using a functional language is to hide the direct manipulation of program states from the programmers. The main advantage of the functional programs in the context of WSN is that it is straightforward to extract parallelism from their manipulation of data. For instance, a function that combines data streams from multiple sensor nodes can be compiled into a form that efficiently aggregates each data stream within the network.

Regiment. Regiment [84] is a functional macroprogramming language for sensor networks. The goal of Regiment is to reduce the programming effort in complex sensor network applications. In Regiment, the principal constructs that the programmer manipulates are signals, *e.g.*, a temperature sensor on a given node, which returns a floating-point value, has type Signal (float). A signal can carry primitive values such as integers, byte, etc. or tuples, such as records containing both the light and temperature sensor readings on a given node. The concept of a region is central to Regiment, representing a collection of signals. The programmer uses regions to express interest in a group of nodes with some geographic, logical, or topological relationship, such as all nodes within k radio hops of some *anchor* node. The operations permitted on region streams include fold, which aggregates values across nodes in the region to a particular anchor, and map, which applies a function over all values within a region. Specifically, map requires no interaction between elements, while fold needs to collapse of data to a single physical point. Regiment programs are compiled into TML [85], an intermediate language, and then to nesC code. The main reason for using TML is to bridge between Regiment and nesC/TinyOS with big semantic gap.

3.3.6 Object-oriented Programming

EnviroSuite. EnviroSuite [86] is an object-based programming model for WSNs. This programming abstraction is based on maintaining a unique mapping between object instances and the corresponding environmental elements. Objects are units that encapsulate program data, computation, communication, sensing and actuation. Dynamic object instances are created automatically by the EnviroSuite run-time system when the corresponding external elements are detected and are destroyed when these elements leave the environment. This also leads to the execution of object code at the location of the corresponding physical entity which is ideal for sensing and actuation tasks.

3.3.7 Set-based Programming

This model of programming is based on the mathematical theory of sets. The choice of set-based programming for WSNs is motivated by the fact that a set is a natural way to think about resource abstraction in a WSN, *e.g.*, using phrases such as "a set of sensor nodes" or "a set of sensor readings".
μSETL. μSETL [87] is a programming abstraction for sensor networks based on set theory. Being a node-level programming model, the scope of a set in μSETL is local to the node where the set is defined, and each set is operated on from a node-level perspective. μSETL implements a compiler that translates μSETL programs to node-specific application code, and a run-time system that provides methods for μSETL programs to perform various set operations such as union, intersection and iterating over the members of a set. It also features two special programming constructs, a periodic block and a monitor block, to support event-driven programming in WSNs. The periodic and monitor constructs allow the programmer to trigger execution of handler functions based on timer events.

3.4 Discussion

In this chapter, we have presented the prominent programming paradigms for sensor networks. Table 3.1 compares the programming models discussed in this chapter from different aspects. Principally, all those models are focused on offering a set of programming constructs to simplify and hasten programming in WSNs, and abstract low-level complexities of layers below the application. Some of them target one or more particular application types and design the programming model based on the common requirements in those applications, *e.g.*, DS-Ware is essentially designed for real-time applications. In contrast, nesC is a general-purpose programming model exploitable in various types of applications. This language is also used as a basis for proposing new special-purpose programming models for WSNs, *e.g.*, Regiment and Agila for functional and agent-oriented programming, respectively.

Programming models such as nesC, Protothread and Think are primarily proposed to develop system-level modules, and at the same time they are use to develop application

Table 3.1: Evaluation of programming paradigms for WSNs.

Programming Abstraction	Programming Paradigm	Execution Platform	Programming Target	Reconf. Support
Agilla	Agent-oriented	TinyOS	Middleware	Yes
NesC	Component-based	TinyOS	All Layers	No
OpenCom	Component-based	Indep.	Application	Yes
THINK	Component-based	Indep.	All Layers	Yes
LooCi	Component-based	Indep.	Application	Yes
ProtoThreads	Event-based	Contiki	All Layers	Yes
DS-WARE	Event-based	GlomoSim	Application	No
ATaG	Imperative	SunSPOT	Application	No
Pleiades	Imperative	TinyOS	Application	No
Regiment	Functional	TinyOS	Application	No
EnviroSuite	Object-oriented	TinyOS	Application	No
μSETL	Set-based	Contiki	Application	Yes

modules and third-party libraries. Although this may lead to an efficient integration of application and low-level system libraries, applications become tightly bound to the underlying programming models. Therefore, in heterogeneous settings the programmer is enforced to deal with different programming technologies to assemble the target application. One efficient solution to tackle this problem is to provide a generic programming technique that is able to abstract different types of system software through a platform-specific run-time system.

The other critical issue, in the context of this thesis, is considering the programming model from a dynamic reprogramming viewpoint. Most of the presented approaches such as nesC, Regiment and ATaG rely on the static programming model and the developed modules are firmly bound together. Contiki's run-time system is perhaps the most popular mechanism that supports dynamic reprogramming of applications written by the Protothread programming model. However, in this model the whole application image must be rewritten when updating one module of the application. Component-based programming models such as OpenCom and Think tackle this problem by enabling fine-grained reconfiguration of software components, event though they impose high memory overhead to the sensor nodes.

Consequently, none of these models seem to promise a generic programming solution for WSNs that can be used across a wide range of hardware and software platforms and efficiently support fine-grained modular reconfiguration. This investigation has motivated us to consider a new programming paradigm that supports the primitives required for reprogramming in WSNs and at the same time is not limited to a particular hardware and software setting.

Chapter 4

Sensor Network Reprogramming

Software deployed on WSNs often need to be updated after deployment for various reasons, such as upgrading node software, fixing software bugs, changing network functionality, tuning module parameters, and patching security holes. However, typical WSNs are deployed in environments where physical access to deployed nodes is very difficult and infeasible. Therefore, besides taking care of programming models for WSNs, there is the vital need of enabling over-the-air reprogramming in sensor platforms. With a WSN-specific reprogramming framework, applications can be remotely maintained rather than collecting nodes from the environment and carrying out the reprogramming operation manually. In this chapter we present the main challenges behind realizing reprogramming in WSNs and discuss the main techniques proposed for updating sensor software over-the-air.

4.1 Background

Beyond static deployments in WSNs, applications may be deployed for long periods of time during which the end-user, network, and environment requirements may change. To maintain such deployments, we need to modify or retask the current deployed application with different sets of configuration parameters and/or modules. The need for sensor software updates may originate from a variety of reasons.

Firstly, a deployed WSN may face sporadic faults that were not observable prior to deployment [18]. This is particularly true in an error-prone harsh WSN deployment where sensor nodes are subject to unexpected and unforeseeable events, making it more difficult to observe and fix failures. A debugging mechanism for WSNs not only concerns with detecting bugs, but also requires to carefully address how new patches should be distributed to target nodes and replaced by faulty code.

Second, software deployed on sensor nodes are becoming larger in size at different levels, from low-level hardware drivers, to operating system, middleware services, and application modules. The evolution in WSN software system has been brought about by employing

sophisticated networking algorithms, implementing intricate use cases, and proposing high-level programming models for WSNs. As the size and the number of software modules become larger, the maintenance task also becomes more important. The maintenance includes upgrading operating system libraries, middleware tools, and application services. In order to maintain long-lived and heavy sensor software, we need a reliable mechanism to remotely patch or upgrade software deployed on sensor nodes through the wireless network.

Third, the requirements from network configurations and protocols may change along the application lifespan because of the heterogeneity and distributed nature of most WSN applications [21]. Therefore, due to resource constraints, it is infeasible to proactively load all services supporting heterogeneity into nodes and hence requirement variations are basically satisfied through updating the sensor software. Such heterogeneous applications require the system to be capable of dynamically reconfiguring itself along several different dimensions such as reconfiguring the network routing, loading new functionality onto devices, and offloading functionality as resources dwindle.

Finally, the increasing number of WSN deployments in pervasive environments makes reconfiguration and self-adaptation two vital capabilities, where a sensor application detects internal and external changes to the system, analyzes them, and seamlessly adapts to the new conditions by updating the software functionalities. Reliable and flexible reprogramming techniques become the central part of any self-adaptation framework when the reconfiguration component undergoes replacing an existing software module with an updated one.

The most relevant form of updating sensor software is remote multi-hop reprogramming using wireless medium and forwarding the new code wirelessly to the target nodes [48]. From a performance viewpoint, energy consumption during reprogramming should also be minimized. Reprogramming is typically done multiple times during the application lifetime and transfers much larger volume of data than that transmitted during regular communication of the sensed data. Thus minimizing resource consumption in reprogramming is of vital importance.

4.2 Reprogramming Challenges

Reprogramming sensor networks as a resource-constrained platform poses several new challenges. First, sensor nodes have limited power supply and the amount of energy consumed to accomplish reprogramming may affect the network lifetime. Update cost is defined as the energy required to reprogram the sensor network, including the energy consumed to receive new updates, as well as overhead imposed by loading the updates on sensor nodes. Some of the dominant energy consumer functions include new code transmission, reading from and writing to external memory, and idle listening. Paying much attention to mitigating energy consumption may result in flexibility degradation. Sensor networks reprogramming makes a trade-off in update cost versus flexibility as shown in Figure 4.1. The flexibility refers to the capability of selecting the most appropriate level of reconfiguration and allowing

arbitrary changes to the functionality.

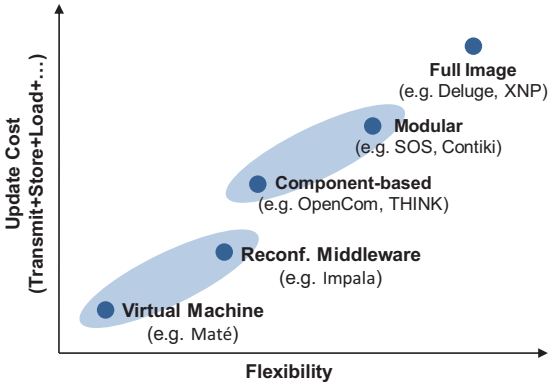

Figure 4.1: Trade-off between flexibility and update cost in WSN reprogramming models.

Second, wireless reprogramming requires 100 percent delivery, which comprises two parts: every node in the network must receive the program code, and the code update image must be received in its entirety. This is in contrast to traditional requirements in sensor network applications, in which, occasional loss of data is tolerable. Moreover, an efficient reprogramming model requires a high communication bandwidth. Unlike sensed data which is small, delivering the entire program image, of the order of kilobytes over low-bandwidth wireless radio, as required in network reprogramming, needs high bandwidth and can lead to energy overconsumption due to packet retransmission.

Third, memory requirements of network reprogramming should be efficiently minimized. Since network reprogramming is supposed to be a service resident on every node of the network, high memory usage would limit the available space for behavior of system-level libraries and application logic. Apart from that, in modular-based reprogramming models memory management becomes more critical as it is required to allocate dynamic memory to the updated modules. This may lead to memory fragmentation and code memory may become divided into many small pieces over several reprogramming.

Finally, the dominant energy consumption happens during new code transmission. Thus, the reprogramming mechanism should minimize the amount of information that needs to be wirelessly transmitted during reprogramming. It means that we need to efficiently delimit and reprogram the necessary portion of sensor software, rather than updating the full binary image.

4.3 WSN Reprogramming Models

WSN reprogramming approaches are distinguished based on above performance challenges. An efficient reprogramming mechanism should be flexible enough to accommodate the new requirements, and at the same time respecting the severe resource constraints imposed during reprogramming. In the rest of this section we report on the prominent frameworks addressing reprogramming in WSNs at different levels, including full software image upgrade, modular upgrades, fine-grained component-based reconfiguration, virtual machines, and parameter updates.

4.3.1 Full Software Image Upgrades

The most popular way to update software in sensor networks is to compile a complete new binary image of the application software together with the system code and overwrite the existing system image of the sensor node. Although full image binary upgrades provide maximum flexibility by allowing arbitrary changes to the functionality, they incur unacceptable update cost by transmitting and uploading large binary images.

Deluge [27] as the most cited approach in this category is a networked bootloader and dissemination protocol that performs full image upgrades of TinyOS applications. Deluge is empowered with an epidemic protocol and operates as a state machine where each node follows a set of local rules to achieve a desired global behavior: the quick, reliable dissemination of large binary objects to many nodes. In order to manage the large size, Deluge divides the binary object into fixed-size pages. Having all pages of a binary image received by a given node, the TOSBoot bootloader reprograms the node by transferring a Deluge object from external flash into program flash. TOSBoot is a static piece of code that executes each time the node exits the reset state. MNP [49] is very similar to Deluge using a sender selection mechanism that attempts to explicitly limit the number of nodes which are transmitting data in a particular broadcast neighborhood. In contrast to the MNP protocol, Deluge requires that radio is always on during reprogramming.

XNP [88] is also a mechanism that implements over-the-air reprogramming of the sensor nodes running TinyOS. It supports basic functions of code distribution and reprogramming, while it does not support multi-hop code delivery. XNP consists of three main components: the network programming module, the bootloader, and the network programming host tool. It stores a new image for the node into an external flash memory, then reads into program memory, and finally reboots the node. MOAP [89] supports multi-hop network reprogramming by disseminating code in a hop-by-hop fashion, that is, a node has to receive the entire program image before starting advertising. To reduce the number of senders MOAP uses a simple publish-subscribe interface.

The above approaches generally come with two main drawbacks: *i)* the large size of the monolithic binary image incurs a high energy overhead for code transmission, *ii)* reinstalling the full application disrupts the running application. Additionally, often a small update in the code, such as a bug fixing, will cause only minor differences between the new and

old system image. Therefore, *difference-based* approaches are proposed to overcome above issues. These are based on sending a diff of the new image into the network as proposed in [90] and [91]. This reduces the amount of data that needs to be transferred. The main drawback of such approaches is that complex algorithms to patch the diff images needs to execute on the sensor nodes.

4.3.2 Modular Upgrades

Systems that support modular upgrades consist of a run-time *loader* and *linker*. The loader is responsible for tracking the storage of the binary modules in the code memory and allocating appropriate resources for them to execute. The linker is in charge of resolving any references made by the modules to the kernel, common libraries, or other modules in the system.

SOS [15] and Contiki [16] are the operating systems allowing modular binary upgrades at run-time. SOS consists of a thin kernel that is statically installed on all the nodes in the network. The rest of the system, including system functionality and application components are implemented as modules. The kernel provides support for loading and unloading modules at run-time, in addition to a rich set of services such as dynamic memory allocation, software timers, sensor manager and high-level I/O interface. SOS uses position independent code to link and load new modules. Position independent code is a type of machine code which does not contain any absolute addresses to itself, but only relative references.

Contiki enables modular updates through in-place dynamic run-time linking and loading of native code using the Executable and Linkable Format (ELF) file format. ELF is not only a standard binary object format on many operating systems for PC computers and workstations, but also feasible even for resource-constrained embedded systems. Using standard ELF format, dynamic linker in the Contiki can dynamically link, relocate, and load updated modules. Compared to SOS, Conitiki's mechanism is independent of microprocessor architectures on the sensor nodes.

4.3.3 Component-based Reconfiguration

Componentization offers the capability of black-box integration of system modules in order to simplify modification and reconfigurability of dynamic systems. This abstraction simplifies reconfiguration realization by formulating the way through which software modules can interact. In particular, software components interact with their environment (*i.e.*, other components) exclusively through interfaces and receptacles. Therefore, to reconfigure a component, we need to maintain the dynamic representation of components topology at run-time and update a component based on meta-data specifying rules and constraints for reconfiguration execution. In general, four types of reconfigurations are likely to happen during the application run-time: *i)* replacing a component with a new one, *ii)* adding a new component, *iii)* component removal, and *iv)* changing the values of component member

variables. In the rest of this chapter we review the existing component-based reprogramming approaches in WSNs.

FlexCup [22] is a code update mechanism that enables on-the-fly reinstallation of software components in TinyOS-based sensor nodes in an efficient way. FlexCup proposes two main phases to reconfigure nesC components: the code generation phase, where relevant information is generated at compile time; and the linking phase, where the modified components are combined with other components at run-time. Specifically, during the code generation process, FlexCup generates meta-data that describes the compiled components. Next, it uses this meta-data during a code update to place the new component inside the running application, relink function calls to the appropriate locations and perform address binding of data objects.

OpenCom [28] also provides run-time reconfiguration programming platform as it is a reflective component model. A reflective model in general presents the internals of the system and provides the means for modifying them [92]. In the case of OpenCom, the internals of the system are represented by the component graph which holds all information about deployed components and how they are connected. By modifying the system graph we can modify the application represented by the graph. Component reconfiguration in OpenCom is supported by the OpenCom run-time. Gridkit [74] is an OpenCom-based middleware for sensor networks, realizing co-ordinated distributed reconfigurations based on policies and context information provided by a context engine.

FiGaRo [25] is a WSN-specific reconfiguration solution, addressing what and where should be reconfigured. FiGaRo mainly focuses on a code distribution algorithm for WSNs. LooCI [78] is a WSN-specific component model providing a loosely-coupled component infrastructure focusing on an event-based binding model for WSNs. LooCI has been successfully implemented on the SunSPOT sensor node. Finally, Think [31] is a C implementation of Fractal [76] whose main goal is to provide fine-grained reconfiguration at architecture-level. Think allows to finely control the overhead induced by meta-data generation required only by the reconfigurable artefacts [77].

4.3.4 Virtual Machines

Virtual machines provide a run-time environment that isolates the execution of applications from the underlying platform. The execution engine proposes a set of high level instruction set, enabling a compact representation of the application code. Therefore, size of applications is reduced in this way and software updates can be distributed more easily. However, since execution takes place within a virtual machine, execution cost is higher compared to native code.

Maté [93] is perhaps the first virtual machine architecture proposed for the resource constrained sensor nodes. Maté runs on top of TinyOS operating system, allowing adding and removing of applications. In Maté and other VM-based approaches, the implementation of the VM is strongly bound to application specifications.

4.3.5 Reconfiguration Middleware

In this approach, dedicated middleware platforms are designed for networked sensor systems, with abstractions that can offer consistent and general mechanisms to configure, deploy, and dynamically reconfigure both system and application level software. The generality of middleware models makes it possible to reuse the reconfiguration functionality on a divers set of dynamic sensor applications.

Impala [94], built as part of the ZebraNet project [95], is a middleware architecture that enables application modularity, adaptively, and reparability in WSNs. Impala proposes a run-time system that acts as a lightweight event and device manager for each mobile wireless sensor node in the system. Moreover, Impala provides an interface for on-the-fly application reconfiguration in order to improve the performance, energy-efficiency, and reliability of the software system. This middleware solution is also potentially able to be exploited in a range of distributed and "grid" computing environments, with broad applicability.

Costa et al. in [21] propose RUNES to provide primary services needed in a typical resource-limited node. Specifically, their work supports customizable component-based middleware services that can be tailored for particular embedded systems. RUNES mainly focuses on Unix-based and Java-based platform, and its implementation on WSNs basically relies on the OS's dynamic facilities, *e.g.*, in [21] Contiki's module update model is exploited.

4.4 Discussion

In this chapter, we have discussed approaches to enable dynamic update of sensor software. The presented models are basically different in the reprogramming scope, which is either whole software image or some portion of software. The former ensures a high level of flexibility, while the latter emphasizes on the efficiency of reprogramming. In the full-image technique, the main concern is to devise an energy-efficient code distribution model to successfully deliver all pieces of code to all nodes in the network. This technique can be an appropriate solution for applications in which reconfiguration rarely happens. In the case of frequent reprogramming, the full-image update model may impose a high energy overhead as it requires to propagate a high volume of new code across the network. The other drawback of this approach is that the context of application (*e.g.*, value of modules' parameters) cannot be preserved during the transition of old image to the updated image.

Modular- or component-based reprogramming imposes a different set of challenges, while they are focused on minimizing the reprogramming cost. Dynamic memory allocation is one of the main shortcomings of approaches discussed in this chapter. For example, Contiki allocates a given space in the memory to load the new module, while component-based models like OpenCom rely on the underlying memory management model of the operating system. The meta-data required to retain information such as functions' address

and bindings between modules (to perform dynamic linking and loading) are the other important issues in this context. Most of existing approaches such as Contiki and SOS maintain a high volume of such data in the data memory. The applicability of approaches in this category over different CPU architectures is the other challenging issue. For example, SOS can be applied on platforms supporting position independent code. The above issues have motivated our research on a component-based reprogramming model for WSNs, while the existing approaches in this area are immature and/or suffer from the extensive use of WSN resources.

Chapter 5

Sensor Service Distribution

Application environments, hosting WSNs, are becoming increasingly heterogeneous at two levels. At a primary level, the sensor nodes in the network may possess varying sensing, processing and communications capabilities due to the efficiency reasons or the nature of applications. At one higher level, WSNs are characterized as a primary element of distributed applications in ubiquitous environments, where sensor nodes populate with actuators, embedded devices, appliances, RFID readers, and mobile devices for monitoring ambient environments and reacting to the external stimuli gathered by different devices. These heterogeneity models require a flexible and simple solution that supports multiple interaction mechanisms and considers the restricted capabilities of sensor devices. In particular, the immaturity of high-level communication protocols and resource scarceness in WSNs bring a critical challenge to the system: how to connect sensor nodes to other computing devices (*e.g.*, mobile devices and actuators) through a standard high-level distribution model. In this chapter, we study this issue by providing an overall background on sensor service integration and distribution, and presenting the state-of-the-art techniques in this area.

5.1 Background

Today, WSN applications are neither able to be developed only on homogeneous constructions, nor focused on carrying out only trivial, limited sensing logics. Sensor network are becoming increasingly heterogeneous due to two primary reasons. First, to have a better throughput with a high deployment density, WSNs are typically organized as a mix of powerful, expensive devices (to perform complex operations) and regular resource-constrained, cheap sensor nodes enabling higher deployment densities and increasing network lifetime. Moreover, during the lifetime of a long-lived WSN, new devices may be developed and integrated to the network, resulting in network heterogeneity. Second, a higher level of nonuniformity occurs when sensor network platforms are integrated with different types of network systems and computing devices in order to realize use cases of ubiquitous ap-

plications. Especially, in context-aware application scenarios, sensor nodes as a primary context data provider element tightly interact with actuators, mobile devices, and other embedded devices.

In such environments, the main focus is on how to abstract the low-level distribution issues and create a uniform interaction model that can be applied for different degrees of heterogeneity, from communications between incongruous sensor nodes, to integration of WSN with modern IT systems, such as the Internet. In this way, sensor nodes become service-enabled devices abstracting their low-level functionality and exposing themselves as a set of well-defined services accessible by other nodes in the network. In the rest of this chapter, we investigate the existing approaches on providing high-level distribution and integration models for WSNs.

5.2 Distributed Callback Functions

This approach is focused on providing low-level APIs to enable implementation of callback-like functions within sensor networks for exchanging raw sensed data among nodes and therefore simplify interaction between distributed modules. The callback function is a reference to an application level function that is passed as an argument to communication libraries within the operating system. This allows the lower-level communication layer to call a function defined in the application layer and pass the data sent by the caller to the function. The communication libraries also consist of a set of proprietary interfaces in order to unicast or multicast plain text messages across the network.

Using callback distribution model, the pure data-oriented nature of interactions in WSNs is promoted to a function-oriented model in which modules running on different nodes can request services from each other through their callback functions. Although this communication model can improve the way to design and implement service-based use cases, the programmer still needs to deal with some low-level issues (*e.g.*, routing protocol) and the applicability of this model is limited to a homogeneous setting over a given network protocol.

5.2.1 Active Message

Active Message [96] essentially refers to the concept of integrating communication and computation, as well as matching communication primitives to hardware capabilities. It is basically proposed to abstract the underlying radio communications services in TinyOS. Active Messages is a simple paradigm for *message-based communication* extensively used in parallel and distributed systems. Each Active Message contains the name of a user-level *handler* to be called on a target sensor node upon arrival and a data payload to pass in as arguments. The handler function is designed to extract the message from the network, as well as either to integrate the data into the computation or to send a response message. The key strengh point of Active Message is that invocation model in handler is inspired

from the event-driven architecture. This allows the programmer to avoid busy-waiting for data to arrive and enables the system to handle communication simultaneously with other activities such as interacting with sensors or executing other application logics.

Active Message takes into consideration the setting in which a large variety of devices with different physical communication capabilities may be deployed. It addresses this issue by building the communication kernel as three separate TinyOS components, then developers can choose which implementations of the basic components is appropriate. Nevertheless, it is still only exploitable for applications that are fully written in the TinyOS programming model and falls short of expectations in heterogeneous deployments.

5.2.2 Chameleon Communication Model

Chameleon [97] is a communication architecture for wireless sensor networks that aims at providing a set of communication primitives that map well onto the communication primitives used by typical sensor network protocols. In particular, it is focused on a communication model that adapts to a wide range of underlying communication mechanisms without requiring any changes to applications or protocols. Chameleon addresses the communications issue in the Contiki operating system. Contiki features a callback invocation mechanism that lies over the Chameleon and provides the primitives to enable easily distributed programming in sensor networks. According to Contiki's callback mechanism, when Chameleon successfully sends a packet (containing data sent by other nodes), it notifies the sending application via a callback. Correspondingly, Chameleon on the data receiving node notifies the associated callback function of application when the data is successfully arrived to the node. Although Chameleon attempts to break the limited communication boundaries to a more diverse set of WSN protocols, it is applicable only in Contiki-based applications.

5.3 RPC-type Invocation

The concept of a Remote Procedure Call (RPC) dates back several decades. A RPC can be described as a mechanism in which applications are able to make calls on remote machines transparently, while for users it appears as local procedure calls. The framework supporting RPCs is in charge of handling underlying RPC complexities, such as converting the calls to a TCP connection between client proxy and server stubs and marshalling/demarshalling the parameters and return values.

The recent implementations of RPC in conventional network platforms include Java RMI [98], CORBA [60], and .NET Remoting [99]. These implementations are designed to support object-oriented and component-based programming models, and to execute on distributed architectures. In the context of WSNs, RPC emphasizes on the need for high-level programming abstractions that hide the complexities of internode communications both within and across single-hop neighborhoods. A number of lightweight component

models have been proposed to provide RPC-like service invocations in sensor networks. We discuss them in this section.

5.3.1 OpenCom

OpenCom [28] is a general purpose component model for building system software without dependency on any platform environment. OpenCOM offers a higher level of abstraction, known as Component Frameworks (CFs), which are used to model interactions between co-operating components. Component bindings in CFs may be local or distributed. OpenCom offers the concept of a distributed component framework to handle interactions between distributed components. Each distributed framework contains a set of local frameworks (local composite components) of the same type. Distributed CPs can be composed of hierarchical local component frameworks and the design of the distributed framework model follows the same basic principles as for local frameworks. The main issue regarding the use of OpenCom in sensor system is its high resource demands. OpenCom has been tried to build components with negligible overhead for supporting features specific to a development area, however it is a generic model and basically developed for resource-rich platforms.

5.3.2 TinyRPC

May et al. in [100] present an RPC extension to nesC [69] which allows RPCs with similar semantics to local calls in nesC. Using TinyRPC, a nesC component can bind to some interface that is actually implemented in a remote node in its neighborhood. This approach consists of three parts. At the high level, it contains a set of nesC language extensions that allow designers to specify module dependencies that span hardware boundaries. It is designed to enable programmers to invoke remote operations, regardless of the underlying communication details. At the middle level, Remote nesC features a set of compiler tools that automate the generation of the communication infrastructure, and transform the language extensions to semantically equivalent nesC code. Finally, Remote nesC provides an operating system service for TinyOS that manages the interactions between remote modules. In this model, developers require to model component interactions on a mote-by-mote basis.

5.4 Web Service Oriented Approach

Service-Oriented Architecture (SOA) offers the potential to provide the necessary system visibility and device interoperability in complex and highly distributed IT systems. It is basically an architectural paradigm that identifies mechanisms to publish, find and compose distributed services. To achieve the above goals, Web Services offer standardized interfaces for loosely coupled software components based on mainly two languages: the Web Service Definition Language (WSDL) and Simple Object Access Protocol (SOAP).

The former is used to define the syntax of the interfaces, and the latter defines the format of messages that are exchanged when invoking services. The Web Services paradigm was initially designed to address a form of distribution that could be used to integrate and combine different computer systems, with a concrete concentration on business architectures. This concentration resulted in relatively heavy standards: both the interface definition (WSDL) and the messages (SOAP) are rather complicated instances of XML documents. Web Services expands the use of the SOA paradigm, implemented through Web Services technologies, in ubiquitous environments, enabling the adoption of a unifying integration technology for all levels of the enterprise, from low-level real-time embedded sensors and actuators devices to enterprise business processes. However, this approach is quite demanding in terms of required computing power, bandwidth and memory usage. In this section, we discuss two initiatives attempted to apply the Web Services paradigm in resource-constrained systems.

5.4.1 SOCRADES

SOCRADES (Service-Oriented Cross-layer Infrastructure for Distributed smart Embedded devices) [32] is a European research project addressing SOA-based manufacturing paradigm. It primarily aims at developing a design, execution and management platform for next-generation industrial automation systems, exploiting SOA paradigm both at the device and at the application level. In particular, it aims at making embedded devices directly accessible across IP-enabled networks with Web Service technologies by installing an adopted Web Service stack.

In SOCRADES, Web service standards are implemented on the physical device level so that all the actors such as sensors, machines, and actuators can exhibit their services through a common interfacing language. The Device Profile for Web Services (DPWS standards) [101] can be used to enable Web service messaging, discovery, description and eventing on resource-constrained devices. However, DPWS is still too heavyweight for the common sensor nodes and needs to be further developed in order to meet the needs of WSNs. Hence, the current SOCRADES framework is only deployable on gateway devices.

5.4.2 Tiny Web Services

The goal of Tiny Web Services [35] is to quantify the resource costs of providing programmatic access to sensor nodes using SOAP-based Web Services. In particular, this approach is focused on implementing web services directly on resource-constrained devices such as sensor nodes using WSDL specification. Tiny Web Services, therefore, eliminates the need for gateway devices that may get tied to a custom format between the gateways and sensor nodes. To this end, the Web Services framework proposes several important guidelines for efficient use of TCP/IP and web services on sensor networks, including the use of persistent TCP connections, disabling delayed acknowledgments in TCP, and using link layer retransmissions. In contrary to traditional web service models assuming an always-on web

services server, Tiny Web Services enables Web Services hosting efficiently on duty cycled nodes that must enter a low power disconnected state for long periods of time.

5.5 RESTful Integration

This model of service distribution is inspired from the principles building the Internet as a scalable global network of computers that interoperate smoothly across heterogeneous hardware and software platforms. These architectural principles as the heart of the Web share a similar goal with the Web Services standards: increase interoperability for a looser coupling between parts of distributed applications. This scalable architectural style is termed as Representational State Transfer (REST), defined by Roy Fielding in his doctoral dissertation [47]. REST emphasizes on scalability of component interactions, generality of interfaces, and independent deployment of components. Moreover, it proposes intermediary components to reduce interaction latency, enforce security, and interoperate with legacy systems. The REST *triangle* defines the principles for encoding (content types), addressing (nouns), and accessing (verbs) a collection of resources using the Internet standards. System, which follows REST principles, is called *RESTful*, *e.g.*, the World Wide Web is the key example of RESTful design.

Resources, which are central to REST, are uniquely addressable using a universal syntax (*e.g.*, a URL in HTTP) and share a uniform interface for the transfer of application states between client and server (*e.g.*, GET/POST/PUT/DELETE in HTTP). REST resources may typically exhibit multiple typed representations using, for example, XML, JSON, YAML, or plain text documents. Thus, RESTful systems are loosely-coupled systems which follow these principles to exchange application states as resource representations. This kind of stateless interactions improves the resources consumption and the scalability of the system.

In a RESTful system, a component can interact with other distributed components by knowing two things: *i)* the unique identifier of the representative resource of component, and *ii)* the predefined standard actions to invoke. Therefore, in this mechanism the application model is transformed from operation-centric into a data-centric one and every data that offers services becomes a resource which is accessible through four standard operations: GET, POST, PUT, and DELETE. In this model of interaction, the client-server separation of concerns can simplify component implementation, reduce the complexity of connector semantics, and increase the scalability of server components.

REST-based Web Services development brings some unique advantages to system, compared to SOAP-based distribution model. Besides the fact that using web patterns makes it more lightweight and simpler, REST uses the Web as an application platform and fully leverages all the inherent features of HTTP such as authentication, authorization, encryption, compression and caching. Traditionally, REST has been used to integrate web systems together. However, the lightweight and simple aspect of REST makes it an ideal candidate for resource-constrained embedded devices to offer services to the digital world.

In this section, we explain the prominent approaches in this area.

5.5.1 TinyREST

TinyREST is one of early attempts to integrate WSNs into the Internet [33]. It uses the HTTP-based REST architecture to obtain/change the state of sensors/actuators. TinyREST assigns a URL address to all resources and uses the POST method to command an actuator to take some action and the GET method to query the current state of a sensor. TinyREST gateway maps a set of HTTP-requests to TinyOS messages in order to link MICA motes to any Internet client. In particular, TinyREST proposes a gateway for communication between WSNs and the Internet through the POST method, GET method, and the REST extensions to HTTP for event subscriptions. Beside the fact that in TinyREST only the access point is able to interact with the Internet (not any individual sensor node), this approach fails to follow all standard HTTP methods.

5.5.2 RESThing

This framework [102] presents an IP-based sensor network system where nodes can directly integrate to modern IT systems through RESTful Web services. This approach relies on the IP protocol stack implemented in the Contiki operating system. Contiki has made a considerable effort on the provision of IPv4 and IPv6 protocols on the common types of sensor nodes. As shown in Figure 5.1, RESThing consists of HTTP Server, REST Engine, SAX based XML parser and Logger modules. Developers can add their own RESTful Web services on the REST engine as symbolized by RWS. RESThing offers an interface to create resources since they are the main abstractions of RESTful Web services.

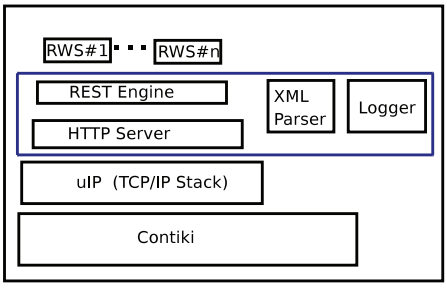

Figure 5.1: Architecture of RESThing framework.

HTTP server is a small footprinted server to handle the incoming and outgoing HTTP requests. It provides an interface to perform certain HTTP related tasks such as accessing request details (headers, entity body, URL path), constructing an HTTP response, etc.

Both REST Engine and SOAP Engine work on top of the HTTP server. The REST framework also includes a XML parser, developed by a third-party [103], to parse requests in XML format. It is very small in code size and being a non-validating SAX based parser makes it memory efficient. A minimal SOAP processing engine is also provided to fulfill SOAP-based Web service invocations. Engine parses the SOAP message using the XML parser, extract the method information and execute it, finally the response SOAP message is built using the XML parser.

5.6 UPnP

The Universal Plug and Play (UPnP) Architecture [104] uses open and standardized protocols based on XML allowing devices to connect seamlessly in corporate environments. In UPnP, all communications are peer-to-peer and transferred over TCP/IP, UDP and HTTP. As all interactions are done atop the IP layer, it is thus completely hardware-independent. In UPnP-based integration, mechanisms for addressing, discovery, description, control, eventing and presentation are defined. Communication in UPnP follows a peer-to-peer philosophy so that no central component is needed to deal with managing interaction among the participants of a UPnP network. This motivates the wide adoption of UPnP as a simple and robust standard for ad hoc and unmanaged networks. UPnP facilitates the integration of new platforms via simple standardized mechanisms because of the capability of zero-configuration and automatic discovery in heterogeneous settings.

5.6.1 A UPnP-based SOA for WSNs

In [34], a UPnP-based approach is presented to enable the integration of WSN platforms in large-scale enterprise environments through a three-layer SOA. This three-layer SOA has been used to integrate with RFID and monitor hazardous chemicals for a petroleum company. The layers of the architecture consist of the backend, gateway, and front-end. The back-end (application) layer consists of Service repository (a database of all available services in WSN), System state manager (keeping track of the states of the sensor nodes), Service mapper (mapping the services to different nodes), Service invocation manager (contacting all the nodes running a given service and returning the results of service invocations to the application), and Notification manager (using a web service to distribute event messages). To exhibit the functionality of a given WSN to the business applications in a uniform way, UPnP gateway is designed to distribute sensor services using the UPnP standard. The gateway takes care of translating between packet-level proprietary sensor network messages and UPnP arguments and assisting in the deployment of services to the sensor network. The front-end (device) layer encompasses the multitude of WSN and radio-frequency identification (RFID) technologies.

5.7 Discussion

Whereas the RPC-based approaches limit the distribution of sensor data and services to homogeneous sensor networks, the Web-service based techniques aim at extending the distribution scope of sensor services to pervasive environments over the IP protocol stack. Although this class of approaches is suitable for a certain range of devices, there is always a class of very small and lightweight devices which will not be able to bear the additional overhead imposed by the Web Service technologies, and therefore sensor networks require a more efficient SOA-based distribution model. Other than that, Web Services development in this approach requires significant expert knowledge and tools that average users do not possess. It makes this framework appropriate for well-defined integration scenarios, but remains quite complicated for ad-hoc integration scenarios by end-users. Although UPnP has mitigated this problem via simple standardized mechanisms, it makes use of some unstandardized protocols like HTTPU. UPnP discovery and control protocols are also heavyweight for a typical sensor node and there is a very limited range of sensor nodes supporting UPnP-based communications. Furthermore, using UPnP in this framework imposes many new hardware and new software set-up for integration support.

REST-based solutions come to tackle the aforementioned problems in SOAP-based Web services. Since most of WSN applications usually offer rather simple and atomic functionalities (*e.g.*, reading sensor values), modeling them using REST is often straightforward. RESThing is an efficient framework to enable REST-based communications in WSNs. However, this framework provides only the primitives required to RESTifying sensor networks. In other words, the programming model to develop web services is missing in RESThing. This motivates our research on proposing a high-level programming model to facilitate REST-based Web service development in WSNs.

Chapter 6

Conclusions and Future Work

This chapter first presents a summary of main results and contributions of this thesis, then some opportunities and ideas for further research are discussed.

6.1 Major Contributions

The main goal of the work presented in this thesis has been to provide support for adaptation and reconfiguration in WSN applications. To achieve that, we have revisited the principles of FCL to devise a reference model addressing adaptation in WSNs. This model is designed to meet all requirements in the lifecycle of a typical dynamic WSN application. These requirements include: *i)* monitoring contextual changes in the environment, *ii)* reasoning about the required adaptation according to the processed context information, and *iii)* reconfiguring the application software based on the decided adaptation. This thesis has contributed in the five main following areas to meet the above requirements:

1. *Context-awareness.* We have proposed an approach for context management in WSN applications. The main idea behind the proposed framework is to provide a high-level abstraction that facilitates observing, modeling and processing WSN-specific context information with respect to the special characteristics of WSNs, *e.g.*, network architecture and context data types. The building block of our proposal is the notion of context node. A context node is context information modeled by a context component performing context execution tasks (context processing, context reasoning, and context configuration). Context components are distributed across the network according to the context model description. The associated run-time middleware system maintains the model and implements a container for context execution.

2. *Adaptation Reasoning.* We have proposed a middleware solution, called WiSeKit, to make an abstraction layer that formalizes and simplifies adaptation reasoning in dynamic WSN applications. The middleware framework is aimed at distributing

the reasoning process across different node types in the network according to the typical hierarchical architecture of WSNs and identifying reasoning models for each hierarchy. The hierarchical adaptation decision is designed based on the resource availability in a node, as well as the portion of the network spanned by a node. In particular, Local-observation happens within sensor nodes to identify local adaptation policies. Intermediate-observation is introduced for gateway nodes and focused on more complicated reasoning required for a region of the network, *e.g.*, a floor or a room in a building. Finally, by Remote-observation, the end-user or sink node would be able to remotely observe the sensor application and decide on the required adaptation for the whole sensor network.

3. *A Programming Model for Adaptive WSNs.* We have proposed a new component-based programming model to simplify application development in WSNs, as well as to support the component-based reconfiguration mechanism proposed in the WiSeKit middleware. This component model, called Remora, addresses high-level event-driven programming in WSNs through a component-based approach. As a SCA-compliant component model, Remora introduces a widely-accepted component programming approach which is specialized for WSNs and embedded systems, and at the same time it attracts PC-based developers to programming in WSNs. To ensure portability of Remora components towards different OSs, the Remora component framework is integrated with the underlying operating system through a well-defined OS-abstraction layer. Since WSN software is inherently event-driven, Remora introduces an efficient way for describing and implementing event-based interactions between software components. This abstraction also aims at simplifying OS-level events processing by translating them to event entities that can be easily integrated to the proposed component model. This component model has been successfully deployed and tested on the TelosB sensor nodes with the Contiki operating system.

4. *Component-based Reprogramming.* This thesis has presented a middleware system, called RemoWare, to address component-based reconfiguration in WSNs. RemoWare includes a set of optimized reconfiguration services deployed on the sensor nodes, which consistently update the required pieces of code. The core contribution of the middleware is to achieve fine-grained reconfiguration in WSNs by revisiting the Remora component model and enhancing it with the principles of *in-situ reconfigurability*. This refers to fine-grained delimitation of static and dynamic parts of sensor software at design-time in order to minimize the overhead of post-deployment updates. To achieve that, RemoWare proposes novel techniques for linking dynamic modules to the system (neighbor-aware binding) and loading the reconfigured components to the memory (in-situ memory allocation). Retaining the state of a component during reconfiguration is the other concern considered by the proposed framework. It also devises an efficient technique for compressing updated components in order to minimize the overhead of radio communications for transmitting

new updates. The RemoWare middleware has been successfully implemented and deployed on the TesloB mote.

5. *Unified Distribution Models.* Distributing WSN services through standard and widely accepted communication protocols is of high importance. Especially, addressing this issue is very critical for the context processing framework in WiSeKit, where sensor nodes may require to communicate with different computing devices to collect context information. This thesis demonstrates a high-level programming abstraction, based on the Remora component model, in order to enable service distribution in WSN applications. This component-based approach promises a new abstraction towards the integration of sensor software modules to the Internet through upgrading component-level services to Web services over a lightweight RESTful architecture. This flexible framework is also potentially able to exhibit sensor services to other types of network protocols by implementing platform-specific bindings.

The Remora component model along with the RemoWare reconfiguration middleware have been made available as open source from the SWISNET project web pages [37]. The intention is to allow others in WSN research community to use these products to develop adaptive applications and validate our results. By allowing others to modify the software, the opportunities for building on our work in future research are also significantly improved.

6.2 Future Work

During the work on this thesis, we have touched on many different problem areas. Although we do not assert the results achieved by this thesis provide a conclusive answer to most of the identified challenges, we believe in the contributions this thesis has made with respect to the state-of-the-art. Therefore, our short-term plan is to further consolidate our solutions by receiving feedback from users of our solutions, and by deploying and evaluating the performance of our framework in real-world scenarios. Additionally, we identify a number of unaddressed issues and potential research directions. In the following, we present open problems and future work for the research challenges discussed in this thesis.

The programming framework we proposed in this thesis is mainly focused on sensor platforms, while it is theoretically portable to other embedded devices used in different application domains, like the ones used in home appliances. Thus, one potential direction for future is to enhance and tailor the Remora component model in order to expand its usage area. Additionally, the current goal of Remora is to be exploited only in application-level programming, while we believe that the efficient support of event processing in Remora potentially enables it to componentize operating systems for WSNs and embedded systems. Such componentization can be even beneficial for developing driver modules for embedded hardware systems.

The code memory allocation model is central to the efficiency of reprogramming model in WSNs as the code memory in most sensor platforms is scarce. Although we have carefully considered this issue in RemoWare, code memory allocation models are considered as an important future research topic for dynamic sensor applications. In particular, avoiding memory fragmentation and improving the reliability during flashing new code are some important research problems in this area. From a network perspective, code distribution is the other interesting issue in WSN reprogramming. *Selective code dissemination* is a recent technique to select target nodes for reprogramming based on various constraints such as sensor node's physical properties or application parameters. We plan to investigate this problem in our future work.

Component-based distributed programming in homogeneous sensor networks is the other open problem. In this thesis, we have studied distribution for heterogeneous networks that are able to accommodate TCP/IP protocol stack, while in a homogeneous setting the distribution model can rely on simple proprietary WSN protocols and reduce the cost of distributed programming by identifying a remote service call to send the required service data from one node to the other node (like Java RMI). Using the concept of component-based programming, the low-level complexities of remote calls can be abstracted by the component run-time system.

In the long term, we foresee that WSN programming and adaption will require stronger abstractions for large-scale distributed applications, in contrast to the existing models (either node-level or network-level approaches) that do not provide a high-level representation of programming and reprogramming features. In particular, in large-scale and heterogeneous applications programmers expect novel programming techniques that enable them to design and implement the target application over a set of virtual nodes that abstract the dense physical deployment. The virtual nodes are represented based on some application-driven logic (*e.g.*, social or physical properties of nodes) and they become the target of programming for the developer, rather than the physical individual nodes.

References

[1] B. Warneke, M. Last, B. Liebowitz, and K. Pister, "Smart dust: communicating with a cubic-millimeter computer," *Computer*, vol. 34, no. 1, pp. 44–51, 2001.

[2] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "A survey on sensor networks," *Communications Magazine, IEEE*, vol. 40, no. 8, pp. 102–114, 2002.

[3] K. Romer and F. Mattern, "The design space of wireless sensor networks," *Wireless Communications, IEEE*, vol. 11, no. 6, pp. 54–61, 2004.

[4] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks*, vol. 2, no. 4, pp. 351–367, 2004.

[5] R. Szewczyk, E. Osterweil, J. Polastre, M. Hamilton, A. Mainwaring, and D. Estrin, "Habitat monitoring with sensor networks," *Commun. ACM*, vol. 47, pp. 34–40, June 2004.

[6] P. Buonadonna, D. Gay, J. Hellerstein, W. Hong, and S. Madden, "Task: sensor network in a box," in *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, 2005, pp. 133 – 144.

[7] J. Burrell, T. Brooke, and R. Beckwith, "Vineyard computing: Sensor networks in agricultural production," *IEEE Pervasive Computing*, vol. 3, pp. 38–45, January 2004.

[8] G. Barrenetxea, F. Ingelrest, G. Schaefer, M. Vetterli, O. Couach, and M. Parlange, "Sensorscope: Out-of-the-box environmental monitoring," in *IPSN '08: Proc. of the 7th Int. Conf. on Information processing in sensor networks*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 332–343.

[9] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh, "Energy-efficient surveillance system using wireless sensor networks," in *MobiSys '04: Proc. of the 2nd Int. Conf. on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2004, pp. 270–283.

[10] A. Milenković, C. Otto, and E. Jovanov, "Wireless sensor networks for personal health monitoring: Issues and an implementation," *Computer Communications (Special issue: Wireless Sensor Networks: Performance, Reliability, Security, and Beyond)*, vol. 29, pp. 2521–2533, 2006.

[11] K. Lorincz, D. J. Malan, T. R. F. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton, "Sensor networks for emergency response: Challenges and opportunities," *IEEE Pervasive Computing*, vol. 3, pp. 16–23, October 2004.

[12] T. Wark, C. Crossman, W. Hu, Y. Guo, P. Valencia, P. Sikka, P. Corke, C. Lee, J. Henshall, K. Prayaga, J. O'Grady, M. Reed, and A. Fisher, "The design and evaluation of a mobile sensor/actuator network for autonomous animal control," in *IPSN '07: Proc. of the 6th Int. Conf. on Information processing in sensor networks*. New York, NY, USA: ACM, 2007, pp. 206–215.

[13] L. Zhang and Z. Wang, "Integration of rfid into wireless sensor networks: Architectures, opportunities and challenging problems," in *GCCW '06: Proc. of the Fifth Int. Conf. on Grid and Cooperative Computing Workshops*. Washington, DC, USA: IEEE Computer Society, 2006, pp. 463–469. [Online]. Available: http://dx.doi.org/10.1109/GCCW.2006.58

[14] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: A survey," *ACM Transactions on Sensor Networks (TOSN)*, vol. 4, no. 2, pp. 1–29, 2008.

[15] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05: Proc. of the 3rd Int. Conf. on Mobile systems, applications, and services*. Seattle, Washington: ACM, 2005, pp. 163–176.

[16] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *SenSys '06: Proc. of the 4th Int. Conf. on Embedded networked sensor systems*. Colorado, USA: ACM, 2006, pp. 15–28.

[17] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: a comprehensive source-level debugger for wireless sensor networks," in *SenSys '07: Proc. of the 5th Int. Conf. on Embedded networked sensor systems*. Sydney, Australia: ACM, 2007, pp. 189–203.

[18] Q. Cao, T. Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks," in *SenSys '08: Proc. of the 6th ACM Conf. on Embedded network sensor systems*. Raleigh, NC, USA: ACM, 2008, pp. 85–98.

[19] A. Taherkordi, R. Rouvoy, Q. Le-Trung, and F. Eliassen, "A self-adaptive context processing framework for wireless sensor networks," in *MidSens '08: Proc. of the 3rd Int. Workshop on Middleware for WSNs*. Leuven, Belgium: ACM, 2008, pp. 7–12.

[20] A. Ranganathan and R. H. Campbell, "A middleware for context-aware agents in ubiquitous computing environments," in *Middleware '03: Proc. of the ACM/IFIP/USENIX 2003 Int. Conf. on Middleware*. Rio de Janeiro, Brazil: Springer-Verlag, 2003, pp. 143–161.

[21] P. e. a. Costa, "The runes middleware for networked embedded systems and its application in a disaster management scenario," in *PERCOM '07: Proc. of the Fifth*

IEEE Int. Conf. on Pervasive Computing and Communications. Washington, DC, USA: IEEE Computer Society, 2007, pp. 69–78.

[22] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," in *EWSN '06: Proc. of the third European Conf. on Wireless Sensor Networks*. Zurich, Switzerland: Springer-Verlag, 2006, pp. 212–227.

[23] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava, "Multi-level software reconfiguration for sensor networks," in *EMSOFT '06: Proc. of the 6th ACM & IEEE Int. Conf. on Embedded software*. Seoul, Korea: ACM, 2006, pp. 112–121.

[24] W. Horré, S. Michiels, W. Joosen, and P. Verbaeten, "Davim: Adaptable middleware for sensor networks," *IEEE Distributed Systems Online*, vol. 9, no. 1, p. 1, 2008.

[25] L. Mottola, G. P. Picco, and A. A. Sheikh, "Figaro: fine-grained software reconfiguration for wireless sensor networks," in *EWSN '08: Proc. of the 5th European Conf. on WSNs*. Bologna, Italy: Springer-Verlag, 2008, pp. 286–304.

[26] B. Porter and G. Coulson, "Lorien: a pure dynamic component-based operating system for wireless sensor networks," in *MidSens '09: Proc. of the 4th Int. Workshop on Middleware Tools, Services and Run-Time Support for WSNs*. Illinois: ACM, 2009, pp. 7–12.

[27] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *SenSys '04: Proc. of the 2nd Int. Conf. on Embedded networked sensor systems*. Baltimore, MD, USA: ACM, 2004, pp. 81–94.

[28] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1–42, 2008.

[29] D. Conan, R. Rouvoy, and L. Seinturier, "Scalable processing of context information with cosmos," in *DAIS '07: Proc. of the 7th IFIP WG 6.1 Int. Conf. on Distributed applications and interoperable systems*. Paphos, Cyprus: Springer-Verlag, 2007, pp. 210–224.

[30] A. Schmidt, *Ubiquitous Computing - Computing in Context*, University of Lancaster, UK, 2002, PhD thesis.

[31] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller, "Think: A software framework for component-based operating system kernels," in *ATEC '02: Proc. of the General Track of the USENIX Annual Technical Conf.* Berkeley, CA, USA: USENIX Association, 2002, pp. 73–86.

[32] L. de Souza, P. Spiess, D. Guinard, M. Khler, S. Karnouskos, and D. Savio, "Socrades: A web service based shop floor integration infrastructure," in *The Internet of Things*, ser. LNCS, vol. 4952. Springer, 2008, pp. 50–67.

[33] T. Luckenbach, P. Gober, K. Kotsopoulos, Andreas Kim, and S. Arbanowski, "Tinyrest: a protocol for integrating sensor networks into the internet," in *REALWSN '05: Proc. of the Workshop on Real-World WSNs*, Stockholm, Sweden, 2005.

[34] M. Marin-Perianu *et al.*, "Decentralized enterprise systems: a multi-platform wireless sensor network approach," *Wireless Communications, IEEE*, vol. 14, no. 6, pp. 57–66, 2007.

[35] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: Design and implementation of interoperable and evolvable sensor networks," in *SenSys '08: Proc. of the 6th ACM Conf. on Embedded Network Sensor Systems*. Raleigh, NC, USA: ACM, 2008, pp. 253–266.

[36] D. E. Comer, D. Gries, M. C. Mulder, A. Tucker, A. J. Turner, and P. R. Young, "Computing as a discipline," *Commun. ACM*, vol. 32, pp. 9–23, 1989.

[37] University of Oslo, "The REMORA Component Model," 2010, http://folk.uio.no/amirhost/remora.

[38] A. Taherkordi, Q. Le-Trung, R. Rouvoy, and F. Eliassen, "WISEKIT: A distributed middleware to support application-level adaptation in sensor networks," in *DAIS '09: Proc. of the 9th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperable Systems*. Lisbon, Portugal: Springer-Verlag, 2009, pp. 44–58.

[39] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. L. Trung, and F. Eliassen, "Programming sensor networks using REMORA component model," in *DCOSS '10: Proc. of the 6th Int. Conf. on Distributed Computing in Sensor Systems*. Santa Barbara, CA, USA: Springer, 2010, pp. 45–62.

[40] A. Taherkordi, R. Rouvoy, and F. Eliassen, "A component-based approach for service distribution in sensor networks," in *MidSens '10: Proc. of the 5th Int. Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*. Bangalore, India: ACM, 2010, pp. 22–28.

[41] D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen, "The digihome service-oriented platform," *Soft. Pract. and Exp.*, 2011.

[42] S. Dobson, S. Denazis, A. Fernández, D. Gaïti, E. Gelenbe, F. Massacci, P. Nixon, F. Saffre, N. Schmidt, and F. Zambonelli, "A survey of autonomic communications," *ACM Trans. Auton. Adapt. Syst.*, vol. 1, pp. 223–259, 2006.

[43] A. e. a. Wood, "Alarm-net: Wireless sensor networks for assisted-living and residential monitoring," *University of Virginia Computer Science Department Technical Report*, 2006.

[44] J. Nehmer, M. Becker, A. Karshmer, and R. Lamm, "Living assistance systems: an ambient intelligence approach," in *ICSE '06: Proc. of the 28th Int. Conf. on Software engineering*. Shanghai, China: ACM, 2006, pp. 43–50.

[45] OSOA, "The service component architecture," http://www.oasis-opencsa.org/sca.

[46] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proc. of the 29th Annual IEEE Int. Conf. on Local Computer Networks*. Tampa, Florida, USA: IEEE Computer Society, 2004, pp. 455–462.

[47] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, USA, 2000, PhD thesis.

[48] P. Levis, N. Patel, D. Culler, and S. Shenker, "Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks," in *NSDI '04: Proc. of the 1st Conf. on Symposium on Networked Systems Design and Implementation*. San Francisco, California: USENIX Association, 2004, pp. 2–2.

[49] S. S. Kulkarni and L. Wang, "Mnp: Multihop network reprogramming service for sensor networks," in *ICDCS '05: Proc. of the Distributed Computing Systems, Int. Conf. on*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 7–16.

[50] B. Pasztor, L. Mottola, C. Mascolo, G. Picco, S. Ellwood, and D. Macdonald, "Selective reprogramming of mobile sensor networks through social community detection," in *EWSN '10: Proc. of the 7th European Conf. on WSNs*, ser. Lecture Notes in Computer Science. Coimbra, Portugal: Springer Berlin / Heidelberg, 2010, vol. 5970, pp. 178–193.

[51] Y. Diao, J. Hellerstein, S. Parekh, R. Griffith, G. Kaiser, and D. Phung, "A control theory foundation for self-managing computing systems," *Selected Areas in Communications, IEEE Journal on*, vol. 23, no. 12, pp. 2213–2222, 2005.

[52] P. Oreizy *et al.*, "An architecture-based approach to self-adaptive software," *IEEE Intelligent Systems*, vol. 14, pp. 54–62, 1999.

[53] IBM Autonomic Computing, "Autonomic computing 8 elements." 2001, http://www.research.ibm.com/autonomic/overview/elements.html.

[54] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, pp. 14:1–14:42, 2009.

[55] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, pp. 46–54, 2004.

[56] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, 2003.

[57] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, pp. 40–48, 2007.

[58] C. Shelton, P. Koopman, and W. Nace, "A framework for scalable analysis and design of system-wide graceful degradation in distributed embedded systems," in *Proc. of the Eighth Int. Workshop on Object-Oriented Real-Time Dependable Systems(WORDS 2003)*, Los Alamitos, CA, 2003, pp. 156–163.

[59] MICROSOFT COM, 1993, www.microsoft.com/com.

[60] CORBA, "Corba component model specifications," 2006, http://www.omg.org/spec/CCM/4.0.

[61] R. Van Ommering, F. Van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[62] J. Muskens, M. Chaudron, and J. Lukkien, "A component framework for consumer electronics middleware," in *Component-Based Software Development for Embedded Systems*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2005, vol. 3778, pp. 164–184.

[63] E. Bondarev, J. Muskens, P. d. With, M. Chaudron, and J. Lukkien, "Predicting real-time properties of component assemblies: A scenario-simulation approach," in *Proc. of the 30th EUROMICRO Conf.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 40–47.

[64] G. Lenzini, A. Tokmakoff, and J. Muskens, "Managing trustworthiness in component-based embedded systems," *Electron. Notes Theor. Comput. Sci.*, vol. 179, pp. 143–155, 2007.

[65] L. Capra, W. Emmerich, and C. Mascolo, "Carisma: Context-aware reflective middleware system for mobile applications," *IEEE Trans. Softw. Eng.*, vol. 29, pp. 929–945, 2003.

[66] K. Geihs, P. Barone, F. Eliassen, J. Floch, R. Fricke, E. Gjorven, S. Hallsteinsen, G. Horn, M. U. Khan, A. Mamelli, G. A. Papadopoulos, N. Paspallis, R. Reichle, and E. Stav, "A comprehensive solution for application-level adaptation," *Softw. Pract. Exper.*, vol. 39, pp. 385–422, 2009.

[67] R. Rouvoy, P. Barone, Y. Ding, F. Eliassen, S. Hallsteinsen, J. Lorenzo, A. Mamelli, and U. Scholz, "Software engineering for self-adaptive systems." Berlin, Heidelberg: Springer-Verlag, 2009, ch. MUSIC: Middleware Support for Self-Adaptation in Ubiquitous and Service-Oriented Environments, pp. 164–182.

[68] C.-L. Fok, G.-C. Roman, and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, 2009.

[69] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *PLDI '03: Proc. of the ACM SIGPLAN 2003 Conf. on Programming language design and implementation*. San Diego, California, USA: ACM, 2003, pp. 1–11.

[70] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, pp. 19:1–19:51, 2011.

[71] Y. Shoham, "Agent-oriented programming," *Artif. Intell.*, vol. 60, pp. 51–92, 1993.

[72] C. Szyperski, *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Boston, MA, USA: Addison-Wesley, 2002.

[73] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Technical concepts of component-based software engineering," Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-2000-TR-008, May 2000.

[74] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes, "Dynamic reconfiguration in sensor middleware," in *MidSens '06: Proc. of the Int. Workshop on Middleware for sensor networks*. Melbourne, Australia: ACM, 2006, pp. 1–6.

[75] GUMSTIX, "Gumstix embedded computing platform specifications," 2004, http://www.gumstix.com.

[76] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[77] F. Loiret, J. Navas, J.-P. Babau, and O. Lobry, "Component-based real-time operating system for embedded applications," in *CBSE '09: Proc. of the 12th Int. Symposium on Component-Based Software Engineering*. East Stroudsburg, PA, USA: Springer-Verlag, 2009, pp. 209–226.

[78] D. Hughes, K. Thoelen, W. Horré, N. Matthys, P. J. del Cid Garcia, S. Michiels, C. Huygens, and W. Joosen, "Looci: A loosely-coupled component infrastructure for networked embedded systems," in *Proc. of the 7th Int. Conf. on Advances in Mobile Computing & Multimedia*. Kuala Lumpur, Malaysia: ACM, Dec. 2009, pp. 195–203.

[79] Sun Microsystems, "Squawk java micro edition virtual machine," 2005, http://squawk.dev.java.net.

[80] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *SenSys '06: Proc. of the 4th Int. Conf. on Embedded networked sensor systems*. Boulder, Colorado, USA: ACM, 2006, pp. 29–42.

[81] S. Li, S. H. Son, and J. A. Stankovic, "Event detection services using data service middleware in distributed sensor networks," in *IPSN '03: Proc. of the 2nd Int. Conf. on Information processing in sensor networks*. Palo Alto, CA, USA: Springer-Verlag, 2003, pp. 502–517.

[82] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner, "The abstract task graph: a methodology for architecture-independent programming of networked sensor systems," in *EESR '05: Proc. of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*. Seattle, Washington: USENIX Association, 2005, pp. 19–24.

[83] N. Kothari, R. Gummadi, T. Millstein, and R. Govindan, "Reliable and efficient programming abstractions for wireless sensor networks," *SIGPLAN Not.*, vol. 42, pp. 200–210, 2007.

[84] R. Newton and M. Welsh, "Region streams: functional macroprogramming for sensor networks," in *DMSN '04: Proc. of the 1st Int. Workshop on Data management for sensor networks: in conjunction with VLDB 2004*. Toronto, Canada: ACM, 2004, pp. 78–87.

[85] ——, "Building up to macroprogramming: an intermediate language for sensor networks," in *IPSN '05: Proc. of the 4th Int. Symposium on Information processing in sensor networks*. Los Angeles, California: IEEE Press, 2005.

[86] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic, "Envirosuite: An environmentally immersive programming framework for sensor networks," *ACM Trans. Embed. Comput. Syst.*, vol. 5, pp. 543–576, 2006.

[87] M. S. Hossain, A. B. M. A. al Islam, M. Kulkarni, and V. Raghunathan, "μSETL: A set based programming abstraction for wireless sensor networks," in *IPSN '11: Proc. of the 10th Int. Conf. on Information Processing in Sensor Networks*, Chicago, Illinois, 2011, pp. 354–365.

[88] Crossbow Tech Inc., "Mote in-network programming user reference," 2003, http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf.

[89] T. Stathopoulos, J. Heidemann, and D. Estrin, "A remote code update mechanism for wireless sensor networks," CENS-TR-30, UCLA, Center for Embedded Networked Computing, Tech. Rep., 2003.

[90] J. Jeong and D. Culler, "Incremental network programming for wireless sensors," in *SECON '04: Proc. of the IEEE Sensor and Ad Hoc Communications and Networks*, Santa Clara, CA, 2004, pp. 25–33.

[91] N. Reijers and K. Langendoen, "Efficient code distribution in wireless sensor networks," in *WSNA '03: Proc. of the 2nd ACM Int. Conf. on Wireless sensor networks and applications*. San Diego, CA, USA: ACM, 2003, pp. 60–67.

[92] F. Kon, F. Costa, G. Blair, and R. H. Campbell, "The case for reflective middleware," *Commun. ACM*, vol. 45, pp. 33–38, 2002.

[93] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *SIGARCH Comput. Archit. News*, vol. 30, pp. 85–95, 2002.

[94] T. Liu and M. Martonosi, "Impala: a middleware system for managing autonomic, parallel sensor systems," in *PPoPP '03: Proc. of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*. San Diego, California, USA: ACM, 2003, pp. 107–118.

[95] ZebraNet Wildlife Tracker, 2002, http://www.princeton.edu/~mrm/zebranet.html.

[96] P. Buonadonna, J. Hill, and D. Culler, "Active message communication for tiny networked sensors," in *INFOCOM '01: Proc. of the 20th Annual Joint Conf. of the IEEE Computer and Communications Societies*. Alaska, USA: IEEE, 2001.

[97] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *SenSys '07: Proc. of the 5th Int. Conf. on Embedded networked sensor systems*. Sydney, Australia: ACM, 2007, pp. 335–349.

[98] Oracle, "Java remote method invocation (java RMI)," 1997-2003, http://www.oracle.com/technetwork/java/javase/tech/index-jsp-136424.html.

[99] Microsoft Corporation, "Microsoft .Net remoting: A technical overview," 2001, http://msdn.microsoft.com/en-us/library/ms973857.aspx.

[100] T. Dean, S. Dunning, and J. Hallstrom, "An rpc design for wireless sensor networks," *Int. Journal of Pervasive Computing and Communications*, vol. 2, no. 4, pp. 384–397, 2007.

[101] OASIS Standard, "Devices Profile for Web Services, Version 1.1," 2009, http://docs.oasis-open.org/wsdd/dpws/1.1/os/wsdd-dpws-1.1-spec-os.html.

[102] D. Yazar and A. Dunkels, "Efficient application integration in ip-based sensor networks," in *BuildSys '09: Proc. of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. Berkeley, CA, USA: ACM, 2009, pp. 43–48.

[103] "Simple xml parser web site." http://simplexml.sourceforge.net.

[104] UPnP Forum, "UPnP Device Architecture 1.0," http://www.upnp.org/resources/documents.asp, Apr. 2008.

Part II

Research Papers

Chapter 7

A Self-Adaptive Context Processing Framework for Wireless Sensor Networks

Authors. Amirhosein Taherkordi, Romain Rouvoy, Quan Le-Trung, and Frank Eliassen

Affiliation. Department of Informatics, University of Oslo, Norway
{amirhost, rouvoy, quanle, frank}@ifi.uio.no

Publication. The Third International Workshop on Middleware for Sensor Networks (MidSens'08), co-located with ACM/IFIP/USENIX Middleware'08, Leuven, Belgium, December 2008.

Abstract. Wireless sensor networks are increasingly being exploited in ubiquitous computing environments as one of the main platforms for gathering context data. In order to continuously observe the environment context during a long period, the sensor node should be considered itself as a context-aware device having particular contextual parameters, such as residual energy or sample rate. Existing work in the field of context-aware computing mostly considers the sensor node as a context data collector agent, regardless of the concern of the node's context elements. In this paper, we first propose an approach for modeling sensor network context information, and then, we introduce a middleware framework that maps our context model to software components, processes the context data, and implements the context model. For this purpose, we propose the notion of *context node*, which is the building block of our context processing framework. The proposed solution is exemplified in the shape of a home monitoring application. Using the proposed framework, the sensor application can adapt itself to the current situation in the environment through

executing a high-level context model describing both the context information to process and the adaptation actions to perform.

7.1 Introduction

Gradually, applications for Wireless Sensor Networks (WSNs) move beyond "sense and send" to pervasive computing environments, where a sensor node has tight interactions with actuators in the environment and behave depending on the context information surrounding it [1, 2]. Applications for such environments must observe continuously their execution context in order to detect the conditions under which some behavioral adaptations are required. This execution context includes various categories of observable entities, such as sensing parameters, residual energy of node, sample rate, or user preferences. Interpretation of context data coming from these entities can be used for improving the execution performance, and adapting application behaviors.

The notion of context has been recognized as an important characteristic of ubiquitous computing environments, where a large number of autonomous agents work together to collect environmental information for smart and interactive devices [3]. Fundamentally, context is defined as "any information about the circumstances, objects, or conditions by which a user is surrounded and that is considered as relevant to the interaction between the user and the ubiquitous computing environment" [4].

According to what has been reported in the literature [5, 6, 7], in most of the context-aware systems, sensor is recognized as a context information provider agent. Depending on the type of possible activities in an environment, various sensor devices should be exploited. The context manager, as a main part of such systems, is in charge of analyzing sensor data and identifying situations where application needs to be adapted. As an example, in a context-aware mobile application, the setting of a mobile device display depends on what is reported from the light detector sensors in the environment. However, the question that arises is whether the sensor node itself should be considered as an adaptable device having its own contextual parameters.

To make this issue more concrete, we consider a monitoring application for which sensor nodes with different capabilities may be deployed in an environment with changing parameters. In this case, we need to extract the different circumstances under which the application is running, process them, and deduce what should be performed in a particular situation. The problem becomes more complex when a lot of conditions come into the play; thereby inserting context management code in the application logic becomes an impractical solution. On the other side, most of the contextual logics are specified by the application user and may change over the application lifespan. Consequently, we face the challenges of *i)* how to model the flow of context information in a WSN application, and *ii)* how this application can be dynamically adapted based on the context model.

In this paper, we propose a context management framework in the middleware layer of WSNs to process context information and provide the necessary analyzed data for

adaptation and reconfiguration tasks. Our work is inspired partially from the COSMOS framework—a comprehensive model for processing context information in ubiquitous computing environments [8, 9].

Specifically in our proposal, each piece of context information is defined as a *context node*. Context nodes can be considered as virtual sensor nodes that are distributed over the physical sensor nodes in the network with respect to the type of information provided by the context node and its role in the context model. Besides, during their deployment, context nodes are mapped to the software components proposed specifically for context data processing in the WSN application. Therefore, the reification of context nodes as components at run-time provides support for the dynamic reconfiguration of sensor nodes whenever their execution context changes.

The rest of paper is organized as follows. In Section 7.2, we demonstrate a motivating application illustrating why context consideration is important in WSNs. The basic concepts of the context middleware are introduced in Section 7.3. The infrastructure of the proposed middleware and implementation issues are presented in Section 7.4. Next, in Section 7.5, the proposal is exemplified based on the motivation scenario described in Section 7.2. We discuss some related work in Section 7.6. Finally, Section 7.7 concludes this paper and identifies some future work.

7.2 Motivating Scenario

As a motivating scenario, let us consider a home-monitoring application. Most of the earlier efforts in this field employed a high-cost wired platform for making the home a smart environment [10, 11]. The emergence of wireless technologies and miniaturized devices make it possible to realize the same functionalities for a smart home more efficient in terms of deployment cost and deployment time. In particular, sensor nodes together with actuators are relevant technologies capable of monitoring various parameters, reasoning about the situation, and reacting to the processed information accordingly [1]. Thus, future home-monitoring applications are expected to be filled with different types of sensors that can provide various context information related to themselves, the inhabitants, and other appliances, such as occupancy, activity, and resource availability. These row of context data are fed to the context-aware home automation system.

Figure 7.1 illustrates an hypothetical context-aware home. Rooms in the home are equipped with various sensor nodes. As shown, depending on the activities that can happen the relevant sensors are deployed, *e.g.*, in the living room three "occupancy" sensors detect movement, two sensors sense the temperature of the room, and one smoke detector sensor determines if a fire is present in the room. Obviously, sensor types and their placement in each room are determined according to the inhabitants' requests, potential home disasters, and technical issues.

The tight interaction of the sensor nodes and the environment on one side, and the inherent resource limitations of WSNs on the other side, complicate requirements analysis

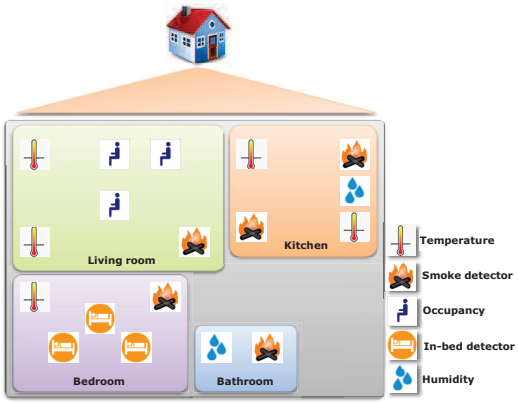

Figure 7.1: Description of the home monitoring system.

and their relations. In this way, adding contextual conditions to the application software in a hardcoded manner becomes an impractical solution. An important question arises on how we can model such information and build a reconfigurable application based on the model. For this purpose, at first, we describe the basic concepts used in building a context model; next the approach for modeling context data will be explained.

7.3 Concepts Of A Context Middleware

Inspired by COSMOS [8], we introduce the architecture of context management in this section. Subsequently, the related concepts are explained. Figure 7.2 illustrates the overall architecture of a context information management framework. This architecture is divided into three layers: the Context Collector, Context Processing, and Context Adaptation layers. Similarly to COSMOS, each layer is organized into a 3-steps cycle of data collection, data interpretation, and situation identification.

The lower layer defines the notion of a Context Collector. Context collectors are software entities that provide raw data about the environment and sensor resources status. The Context collector also encompasses information coming from user preferences. The rationale for this choice is that context collectors should provide all the inputs needed to reason about the execution context. In our hierarchical architecture for WSN, the responsibility of context collectors is assigned to the sensors.

The middle layer defines the notion of Context Processing. Context processors filter and aggregate raw data coming from context collectors. The purpose is to compute some high-level, numerical or discrete, information about the execution environment. Data provided by context processors are fed into the adaptation layer. Intuitively, context processing

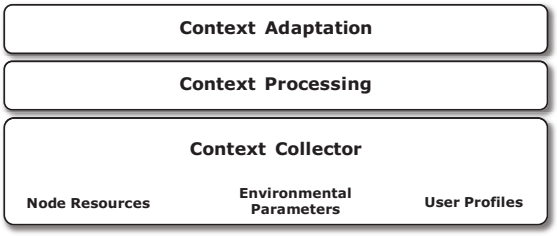

Figure 7.2: Architecture of a context management framework.

tasks should be performed by the intermediate more powerful nodes in the hierarchical WSN architecture, namely, cluster heads and sink node.

The upper layer is concerned with the process of decision making. The purpose is to be able to make a decision on whether or not an adaptation action should be performed. The Context Adaptation layer is thus a service that is provided to applications and that encapsulates the situations identified by context nodes and processors. For example, in the motivating scenario, changing the behavior of temperature sensor in case of room occupancy is a decision made within this layer.

7.3.1 Architecture of a Context Node

The above architectural model defines the main aspects of a typical context management system. To exploit this model for WSN applications, each layer needs to be tailored according to the limitations and specifications of WSNs. In particular, for each layer it is necessary to extract the tasks and find the appropriate node in the network for performing these tasks.

We define the notion of *context node* as a representative of functionality in the context management architecture. In fact, a context node is the basic structuring of the architecture. A context node reifies particular context information. Context nodes are organized into hierarchies, which are compatible with hierarchies defined in WSN architectures. The graph of context nodes represents the set of context management policies associated to the application logic.

Thus, as illustrated in Figure 7.3, a context node interacts with other context nodes by exchanging messages, which encapsulate context information reports and are handled by the *message manager*. The context node can be either active or passive. A passive node obtains messages upon demand, while an active node gathers periodically messages via the *activity manager*. The *context processor* is responsible for processing the received messages into context information of higher-level of abstraction and can, eventually, operate some functional or non-functional actions on the enclosed context nodes. These actions are planned and executed by a *context reasoner* and a *context configurator*, respectively. The

context model we define supports the sharing of *Context Reasoner* and *Configurator* as well as *Activity* and *Message Managers* across collocated context nodes in order to reduce the memory footprint and the resource consumption of the sensor nodes.

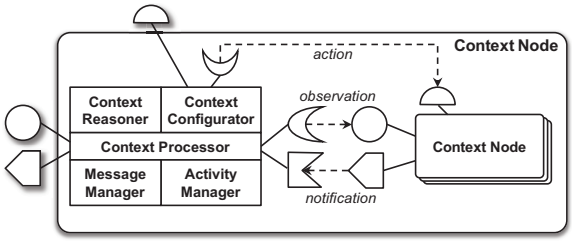

Figure 7.3: Architecture of a context node.

As an example from the motivating scenario, the temperature sensors deployed in the room enclose four context nodes (cf. Figure 7.4). At the leaf level, a context node encapsulates the hardware sensor and converts raw data into the context information reports. This report covers not only temperature reports, but also clock and energy information. The current temperature and the energy left are extracted by two other context nodes (meaning that a context node can be shared among other context nodes). The energy status is then processed by a fourth context node in order to be compared to a given threshold that determines if the action of reconfiguring the sample rate should be taken or not. The current temperature is notified to the context node of the parent domain (the room is this case) in order to be processed.

7.3.2 Composition of Context Nodes

Each context node is placed in one of the layers of the context processing architecture based on its responsibility. The context model we define organizes the collaboration between context nodes.

Figure 7.4 illustrates the context model of a dynamic home-monitoring system. Hardware sensors deployed in the rooms produce context information that is continuously processed to identify potential *actions* to execute. These actions can be functional (*e.g.*, activation of the alarm bell) or non-functional (*e.g.*, reconfiguration of a sensor sample rate). The context information is propagated within the monitoring system proactively or reactively via *observation* and *notification* mechanisms, respectively. *Context domains* identify context nodes that can be collocated in order to reduce the cost of communication and thus react autonomously (without relying on a centralized architecture).

A context node in the context model can be shared between other nodes. The sharing of a context node corresponds to the sharing of a context management policy. Nodes in the lower level of the hierarchy are more likely to be shared. Such nodes are more expected

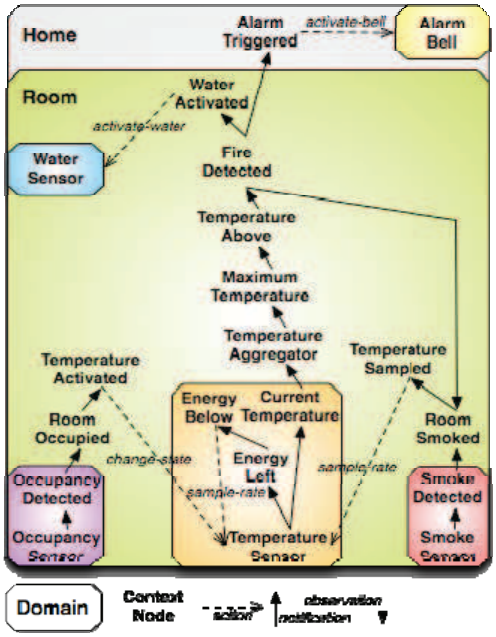

Figure 7.4: Context model of the monitoring system.

to collect context data, rather than performing any action. For instance, the context node providing raw information about the temperature—*i.e.*, Temperature Sensor—falls in this category.

Our proposal for context model is more in accordance with hierarchical architectures for WSNs. The frequent reference to this model in WSN challenges makes the proposed context model a reference model for addressing context management issues for WSNs. Note that like hierarchical WSNs architectures, our context model does not impose any limitation on the depth of the hierarchy.

In addition to the propagation of context information, the context nodes can embed some autonomic behavior to react to context changes. In fact, context nodes basically provide the raw data for identifying the type of adaptation required. Like the design model of the context node, the adaptation model is expected to follow the component model in order to have a unified approach for addressing self-* issues. This feature enables the context model to be self-adaptive and thus adapts the context information retrieval characteristics depending on the context currently manipulated.

7.4 Implementation Of A Context Middleware

In this section, we put the concepts mentioned in the previous section all together and describe the cycle of context management first. Next, the infrastructure of a middleware facilitating the context management tasks will be presented.

Context management encompasses all activities required to reach a context-aware application. The major phases of management can be categorized as *i)* classification of context information for a particular computing environment, *ii)* finding the relation between the classified information, *iii)* extraction of context elements (context nodes in our scope), *iv)* putting context elements together in an interaction model, *v)* mapping the context model to the platform supporting context elements definition and interaction, and *vi)* iterating these steps for the possible future needs.

Context management frameworks have tried to span as much as possible of the above tasks, although the main focus has been on the modeling. Apart from concerns directly related to each step, the underlying platform brings its own challenges and concerns. The limitations and characteristics of a particular platform specialize some steps of context management. Particularly, context modeling and execution need to be considered precisely with respect to the target platform.

Moreover, frameworks for context execution have been a challenging issue in terms of the degree of generality, implementation techniques, and non-functional features, such as scalability. Proposals target middleware layer solutions to address such concerns. A middleware solution can simplify the task of maintaining context-aware systems and provide common services for context management. It supports a common model for context data definition and also it is able to maintain the context model dynamically.

Our proposal relies on a component-based middleware for providing context information in WSN applications. Thus, we have investigated several component models [12, 13, 14] to identify features suitable for our context model. Based on this we extract a fundamental characteristic of state-of-the-art component model: A *lightweight hierarchical* component model that stresses on *modularity* and *extensibility*. It allows the definition, configuration, dynamic reconfiguration, and clear separation of functional and non-functional concerns. The central role is played by interfaces, which can be either *functional* or *control*. Whereas functional interfaces provide external access point to components, control interfaces are in charge of non-functional properties of the component (*e.g.*, life-cycle management or binding management). Components are sometimes divided into *passive* and *active*. Whereas passive components generally represent services, active components contain their own thread of control.

Based on these assumptions, a context node is implemented as a composite component that contains at least five primitive components: the context processor, the context reasoner, the context configurator, the activity manager, and the message manager. The context node's dependencies are enclosed as components as well. The observation and notification mechanisms are implemented by functional interfaces, while the support for

reconfiguration is provided by the control interfaces.

The middleware run-time system is composed of two major parts: core services and core context components. Core services are dedicated for maintaining the context model, *e.g.*, *communication service* for making the interaction of context nodes located in the different nodes. There are also some context nodes that are frequently used in the context models, *e.g.*, ResidualEnergy. The middleware is equipped with such core context components as well.

Moreover, the middleware is in charge of maintaining the context model. Particularly, the middleware run-time system takes care of managing context nodes and their interactions according to the context model description. Figure 7.5 illustrates how the context model is mapped to the context components. Each context component represents a portion of context model (one context node or more context nodes and their interactions). The logic behind the context model specifies which portion of model should be mapped to a particular context component. Based on this knowledge, the middleware will be able to deploy context components over the sensor nodes, handle local and remote interactions of context components, and provide the run-time system for context-aware application execution.

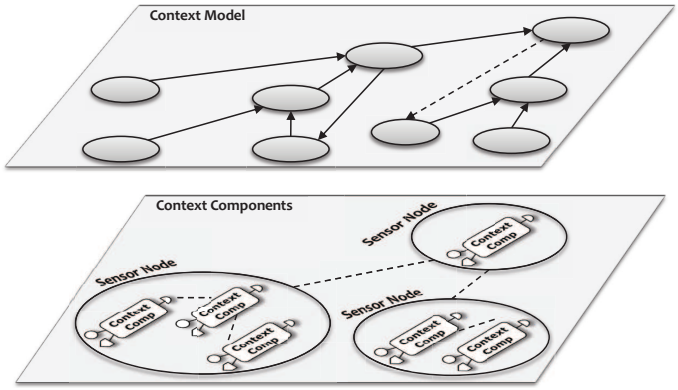

Figure 7.5: Mapping of the context model to the context components.

7.5 Sample Scenario Execution

Let us again refer to the sample home-monitoring application and consider how the context components and their interactions can be obtained from the context model. As mentioned before, we propose the context domain for identifying collocated context nodes in order to process information locally. In fact, each context domain represents a physical node in the

system. The most inner domains represent the sensor nodes and domains encompassing them are cluster heads, while the outer domain surrounds all clusters context information.

Figure 7.6 presents the context components and their interactions for a temperature sensor, occupancy sensor, and cluster head. The `TemperatureSensor` component provides two kinds of contextual information: *current temperature* and *residual energy* of the sensor node (see 7.4). For the former one, a context component reading the current environment temperature is deployed. All temperature sensors inside the room send the output of this component to the temperature aggregator component located in the cluster head. The latter information is provided by the `EnergyLeft` component. In case of reaching a threshold of residual energy, the `EnergyBelow` component calls the `sampleRateControl` function from `TemperatureSensor` to reduce the sample rate. `OccupancySensor` component running on the occupancy sensors notifies the `RoomOccupied` component in the cluster head in case of detecting any movement. `TemperatureActivation` component detects room occupancy. Upon receiving any notification from `RoomOccupied` component it calls the `changeState` function of `TemperatureSensors` and makes them silent while receiving occupancy notification.

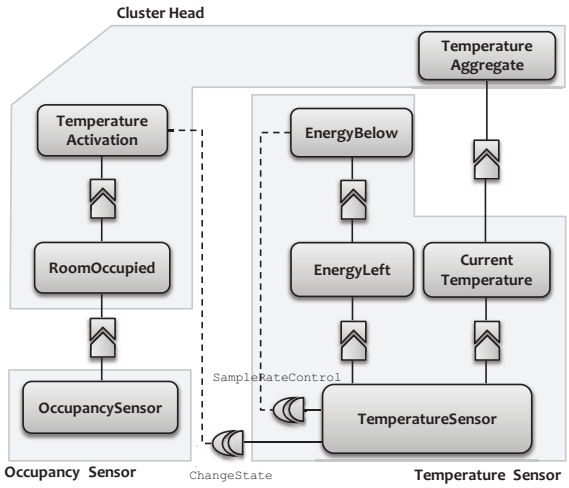

Figure 7.6: Context components composition for home monitoring.

The interactions of components located in a node are established via local message passing. As of context node architecture, the message manager in each component receives the context information, processes it, and delivers the result to the context processor part. The context components interacting directly to each other are expected to know the format of message passed among them. Likewise, components running in different nodes interact via message passing, however, in this case messages are sent over the data distribution

protocol of network.

The middleware run-time system is in charge of connecting context components via message passing according to the context model description. The middleware *communication service* also encapsulates the remote message and sends it to the relevant node within the network. Similarly, in the target node, this service reads the message and delivers it to the relevant context component for further processing.

Figure 7.6 presents just a small part of managing the home context information. In fact, this model is part of a larger model that encompasses all context elements. The middleware supporting such model should be able to manage the context model in a scalable fashion. Maintaining the context model in a particular sensor node needs just the *meta-context information* related to that node including *i)* the context components that should be deployed within that node, and *ii)* the details of local and remote interactions of those components. In our proposal, the middleware in each sensor node is equipped with such metadata. In this way, the large amount of meta-context information is split among sensor nodes according to their role in the context model execution.

7.6 Related Work

As mentioned at the beginning of this paper, most of the context processing work employs the sensor node just as a context data collector, whereas the concern of context awareness for WSN application is a separate topic indicating how the WSN itself can be managed based on its own context data.

The first relevant work has been reported by Huebscher et al. [6]. They propose an adaptive middleware for context-aware application that abstract the applications from the sensors that provide context data. The proposed middleware is a layered architecture for context provision. In this architecture sensors are placed at the bottom as raw context data provider. In the one upper level, a network of context services is located. The main contribution of this work is to choose the best set of available context services in order to satisfy the context-aware application requirements. The motivation for this work is a home automation application for smart appliances in the home equipped with sensors. The sensors in such a smart-home are not necessarily networked, but play as an isolated agent for gathering the context data. However, in our proposal a sensor is characterized as an autonomous node of a network capable to adapt itself against the environmental and sensory context data.

Some work uses the context-aware concept as a technique for conserving energy in WSNs. [5] proposes a framework for supporting the use of context to trigger power saving functionalities in the sensors—*i.e.*, controlling the behavior of the sensors. For this purpose, they propose *context discovery* to discover useful contexts from sensor data, and *context-trigger engine* to use the discovered context as a trigger. In this work, two centralized databases are proposed for indentifying context information and context actions. This work is suitable for applications with simple isolated context information. However, the overhead

of centralized observation method in this framework may compromise the improvement claimed in the context-based energy conservation.

TeenyLime is the recent reported middleware solution for dealing with the complexity of specifying how multiple tasks coordinate to accomplish a functionality in the wireless sensor and actor networks [15]. TeenyLIME operates by distributing the tuple space among the devices, transiently sharing the tuple spaces contents as connectivity allows, and describing reactive operations that fire when data matching a template appears in the tuple space. In fact, TeenyLIME system provides a single unified abstraction for representing both the application and system context. TeenyLime can takes part in our model as a context processor if the message format, adopted for context nodes interactions, conforms to the tuple structure.

7.7 Conclusions And Future Work

Using WSNs in autonomous environments makes the application design process more challenging. This process is based on gathering, analyzing and treating large amount of context data produced by the environment and sensor nodes, and then adapting and configuring application based on this data.

In this paper, we presented an approach for addressing context related concerns in the WSNs. Mainly, our approach is composed of two parts *i)* a context information processing architecture for modeling the sensor context elements and their interactions, and *ii)* a middleware framework for executing the context model. The basic structuring concept of our proposal is the context node. In fact, a context node is context information modeled by a context component performing context execution tasks (context processing, context reasoning, and context configuration). Context components are distributed over the network according to the context model description. The underlying middleware run-time system maintains the model and provides a container for context execution. Using this middleware, a WSN application can adapt itself to the current situation in the environment through executing a high-level context model describing the context information and the adaptation actions corresponding to the context.

We are currently focusing on the home-monitoring application as a motivating scenario. In this paper, we described this application briefly and discussed how its context model can be obtained and also the equivalent context components composition for a part of context model was illustrated at the end of paper.

This work will be continued along two main axes. First, the initial version of middleware framework will be developed based on the requirements mentioned in this paper. The runtime middleware system should be equipped with services for facilitating context data processing, reasoning and configuration, besides the services for supporting context model execution. Second, the middleware system will be evaluated by developing the context-aware home monitoring application. Also, the performance of the middleware will be evaluated in terms of memory occupation, context model execution processing overhead,

and energy consumption.

References

[1] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks*, vol. 2, no. 4, pp. 351–367, 2004.

[2] D. Puccinelli and M. Haenggi, "Wireless sensor networks: applications and challenges of ubiquitous sensing," *Circuits and Systems Mag., IEEE*, vol. 5, no. 3, pp. 19–31, 2005.

[3] S. S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta, "Reconfigurable context-sensitive middleware for pervasive computing," *IEEE Pervasive Computing*, vol. 1, no. 3, pp. 33–40, 2002.

[4] A. Ranganathan and R. H. Campbell, "A middleware for context-aware agents in ubiquitous computing environments," in *Middleware '03: Proc. of the ACM/IFIP/USENIX 2003 Int. Conf. on Middleware*. Rio de Janeiro, Brazil: Springer-Verlag, 2003, pp. 143–161.

[5] S. K. Chong, S. Krishnaswamy, and S. W. Loke, "A context-aware approach to conserving energy in wireless sensor networks," in *PERCOMW '05: Proc. of the Third IEEE Int. Conf. on Pervasive Computing and Communications Workshops*. Washington, DC, USA: IEEE Computer Society, 2005, pp. 401–405.

[6] M. C. Huebscher and J. A. McCann, "Adaptive middleware for context-aware applications in smart-homes," in *MPAC '04: Proc. of the 2nd workshop on Middleware for pervasive and ad-hoc computing*. Toronto, Canada: ACM, 2004, pp. 111–116.

[7] R. Couto Antunes da Rocha and M. Endler, "Middleware: Context management in heterogeneous, evolving ubiquitous environments," *IEEE Distributed Systems Online*, vol. 7, no. 4, p. 1, 2006.

[8] D. Conan, R. Rouvoy, and L. Seinturier, "Scalable processing of context information with cosmos," in *DAIS '07: Proc. of the 7th IFIP WG 6.1 Int. Conf. on Distributed applications and interoperable systems*. Paphos, Cyprus: Springer-Verlag, 2007, pp. 210–224.

[9] R. Rouvoy, D. Conan, and L. Seinturier, "Software architecture patterns for a context-processing middleware framework," *IEEE Distributed Systems Online*, vol. 9, no. 6, p. 1, 2008.

[10] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *WSNA '02: Proc. of the 1st ACM Int. Workshop*

on Wireless sensor networks and applications. Atlanta, Georgia, USA: ACM, 2002, pp. 88–97.

[11] M. C. Mozer, "Lessons from an adaptive home," *Smart Environments: Technology, Protocols, and Applications*, pp. 273–298, 2004.

[12] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[13] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1–42, 2008.

[14] I. Crnkovic, *Building Reliable Component-Based Software Systems*, M. Larsson, Ed. Norwood, MA, USA: Artech House, Inc., 2002.

[15] P. Costa, L. Mottola, A. L. Murphy, and G. P. Picco, "Programming wireless sensor networks with the teenylime middleware," in *Middleware '07: Proc. of the ACM/IFIP/USENIX 2007 Int. Conf. on Middleware*. Newport Beach, California: Springer-Verlag New York, Inc., 2007, pp. 429–449.

Chapter 8

WiSeKit: A Distributed Middleware to Support Application-level Adaptation in Sensor Networks

Authors. Amirhosein Taherkordi (1), Quan Le-Trung (1), Romain Rouvoy (1,2), and Frank Eliassen (1)

Affiliation.

(1) Department of Informatics, University of Oslo, Norway
{amirhost,quanle,rouvoy,frank}@ifi.uio.no

(2) INRIA Lille Nord Europe, ADAM Project-team, University Lille 1, LIFL CNRS UMR 8022, Villeneuve dAscq, France
{romain.rouvoy}@inria.fr

Publication. The 9th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS'09), Lisbon, Portugal, June 2009.

Abstract. Applications for *Wireless Sensor Networks* (WSNs) are being spread to areas in which the contextual parameters modeling the environment are changing over the application lifespan. Whereas *software adaptation* has been identified as an effective approach for addressing context-aware applications, the existing work on WSNs fails to support context-awareness and mostly focuses on developing techniques to reprogram the whole sensor node rather than reconfiguring a particular portion of the sensor application software. Therefore, enabling adaptivity in the higher layers of a WSN architecture such as the

middleware and application layers, beside the consideration in the lower layers, becomes of high importance. In this paper, we propose a distributed component-based middleware approach, named WiSeKit, to enable adaptation and reconfiguration of WSN applications. In particular, this proposal aims at providing an abstraction to facilitate development of adaptive WSN applications. As resource availability is the main concern of WSNs, the preliminary evaluation shows that our middleware approach promises a lightweight, fine-grained and communication-efficient model of application adaptation with a very limited memory and energy overhead.

8.1 Introduction

The challenges for application development on WSNs are becoming as prominent as the issues concerning sensor hardware, network architecture, and system software. It is because the new emerging applications for WSNs do not limit themselves to a single function called *"sense and send"* with trivial local data processing tasks [1, 2]. Applications for WSNs are gradually moving towards *pervasive computing environments*, where sensor nodes have tight interactions with actuators, deal with the dynamic requirements and unpredictable future events, and behave based on the context information surrounding them [3].

In such an environment, in addition to the basic tasks, an application needs to adapt its behavior to cope with changing environmental conditions, and different capabilities of each individual sensor node in the network. As different nodes are expected to run different tasks, software with dynamic adaptable functionalities becomes an essential need. Moreover, for applications deployed to a high number of nodes in inaccessible places, individual software updating becomes an impractical and inefficient solution.

As the common types of sensor nodes are still suffering from resource scarceness, researchers have not been willing to consider the application code on sensor node as adaptive software. This is because, on one hand, the typical adaptation frameworks mostly come with a high level of complexity in the reasoning process and reconfiguration mechanism. On the other hand, most of the software development paradigms for WSN application are not able to support reconfigurability due to the lack of modularity, such as in the case of script programming. Moreover, the lack of operating system level support for dynamic reconfiguration is the other critical challenge in the way of achieving application-level adaptivity in WSNs. Recently, operating systems, such as Contiki [4], have considered this issue by supporting dynamic binding and loading of software components.

A few works have been reported in the literature that address adaptation for embedded and sensor systems. In [5, 6, 7], the main contribution is to provide adaptivity at the middleware-level (not application-level) in order to make the network-level services reconfigurable and replaceable. In [8], a small runtime support is proposed over Contiki to facilitate dynamic reconfiguration of software components. Although it promises to achieve application-level reconfigurability, the level of abstraction is low and it does not propose a general framework supporting all essential aspects of application adaptation. In fact, it

plays the role of component reconfiguration service in a typical adaptation framework.

The performance of the adaptation middleware depends on two major factors. The first is the *reconfigurability degree* of software modules. In a highly reconfigurable model, the update is limited to the part of the code that really needs to be updated instead of updating the whole software image. We term this feature *fine-grained reconfiguration*. The second is the mechanism by which a module is reconfigured. In this paper, we concentrate on the latter, whereas the former has been discussed in [9] by proposing a new component model, called *ReWiSe*, for lightweight software reconfiguration in WSNs.

In this paper, we present a novel distributed middleware approach, named WiSeKit, for addressing the dynamicity of WSN applications. WiSeKit provides an abstract layer accelerating development of adaptive WSN applications. Using this middleware, the developer focuses only on application-level requirements for adaptivity, while the underlying middleware services expose off-the-shelf APIs to formalize the process of adaptive WSN application development and hide the complexity of the technical aspects of adaptation. The adaptation decision logic of WiSeKit is also inspired from the hierarchical architecture of typical WSNs in order to achieve the adaptation goals in a light-weight and efficient manner.

The rest of this paper is organized as follows. In Section 8.2, we demonstrate a motivating application scenario. The basic design concepts of our middleware proposal are described in Section 8.3. In Section 8.4, the WiSeKit middleware is proposed with a preliminary evaluation presented in Section 8.5. Related work is presented in Section 8.6. Finally, Section 8.7 concludes the paper and gives an outlook on future research.

8.2 Motivating Application Scenario

In this section we present an application scenario in the area of *home monitoring* to further motivate our work. Most of the earlier efforts in this field employed a high-cost wired platform for making the home a smart environment [10, 11]. Future home monitoring applications are characterized as being filled with different sensor types to observe various types of ambient context elements such as temperature, smoke, and occupancy. Such information can be used to reason about the situation and interestingly react to the context through actuators [3].

Figure 8.1 illustrates a hypothetical *context-aware home*. Each room is equipped with the relevant sensor nodes according to its attributes and uses. For instance, in the living room three "occupancy" sensors are used to detect the movement, one sensor senses the temperature, and one smoking sensor for detecting the fire in the room. Although each sensor is configured according to the preliminary requirements specified by the end-user, there may happen some predictable or unpredictable scenarios needing behavioral changes in sensor nodes. Basically, these scenarios can be considered from two different aspects: *i)* application-level, and *ii)* sensor-level. The former refers to the contextual changes related to the application itself, *e.g.*, according the end-user requirements for the living room,

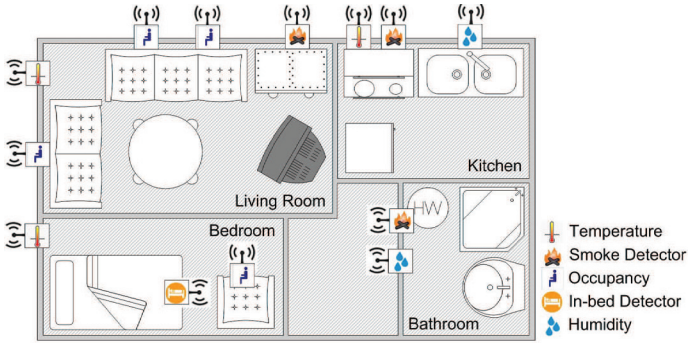

Figure 8.1: Description of the home monitoring system.

if one of the occupancy nodes detects a movement in the room, the temperature nodes should stop sensing and sending tasks. The latter further concerns with the capabilities and limitations of a particular sensor node, *e.g.*, if the residual energy of temperature sensor is lower than a pre-defined threshold, the aggregated data should be delivered instead of sending all individual sensor readings.

Besides the above concerns, the recent requests for *remote home monitoring*, which enables the owner to check periodically home state via a web interface, are being extended by the request of *remote home controlling*. This need also brings some other new challenges in terms of dynamicity and makes the issue of adaptivity more significant.

Considering statically all above concerns becomes quite impossible when you have many of these scenarios that should be supported simultaneously by the application on a resource-limited node. Moreover, at the same time you need to maintain the relation between the context elements and reason timely on a change. Obviously, supporting all these requirements during application run-time needs an abstract middleware approach to address the dynamicity and adaptivity challenges w.r.t. the unique characteristics of WSNs.

8.3 Basic Design Concepts

In this section, we describe the basic design concepts of WiSeKit middleware.

Adaptation Time. Basically, adaptation can be performed in two manners: *statically* and *dynamically*. Static adaptation relates to the redesign and reconfiguration of application architectures and components before deployment of the application. Dynamic adaptation is performed at application run-time due to the changing needs, resource and context conditions. We adopt dynamic adaptation in our proposal because most WSN applications are expected to work seamlessly for a long time and mostly deployed in inaccessible places.

Adaptation Scope. Two popular adaptation mechanisms are introduced in the literature [12], [13]: *parameter adaptation* and *component adaptation*. Parameter adaptation supports fine tuning of applications through the modification of application variables and deployment parameters, while component adaptation allows the modification of service implementation (replacement of component), adding new components, and removing running components. We explain later in this paper why and how our middleware supports both of these mechanisms.

Adaptation Policy. In general, there are three different approaches for identifying a policy: situation-action rules [13, 14], goal-oriented [15] and utility-based [16]. The two latter techniques represent high level forms of policies, while the former specifies exactly what to do in given situations. As the adaptation policies of most WSN applications can be described easily through a set of conditional statements, WiSeKit follows the situation-action rules approach. Situations in our proposal are provided from the framework we proposed in [17]. This framework proposes a *context processing model* to collect data from different sources (environment, application, sensor resources, and user) and process them for the use of adaptation reasoning service.

Fine-grained Reconfiguration. Adaptation reasoning specifies through which mechanism (either parameter-based or component-based) which parts of the application should be reconfigured. As the major cost of reconfiguration in WSNs is in transferring the new update code across the network, fined-grained reconfiguration becomes very important. Note that fine-grained reconfigurability should be supported by the underlying system software.

Hierarchical Adaptation. As the sensor nodes are mostly organized in a hierarchical way [18], our proposal distributes the adaptation tasks among nodes according to the level of hierarchy of a particular node. Hierarchical adaptation is based on the idea of placing adaptation services according to: *i)* the scope of information covered by a particular node, and *ii)* the resource richness of that node.

8.4 WiSeKit Adaptation Middleware

WiSeKit aims to provide a set of APIs at the middleware level of WSNs in order to make an abstraction layer that formalizes and simplifies the development of adaptive WSN applications. In general, WiSeKit is characterized by the following features:

- *Local-adaptivity*: an application running on the sensor nodes has the possibility of identifying its adaptation policies. The APIs exposed at the middleware layer are able to read the policy object and maintain the application components' configuration according to the context information gathered periodically from both sensor node and application.
- *Intermediate-observation*: using WiSeKit, we can specify adaptation requirements for a region of the network, *e.g.*, a floor or a room in a building. At this level, we

can specify high-level adaptation policies through WiSeKit APIs provided at more powerful nodes such as cluster head or sink node.

– *Remote-observation*: the end-user or the agent checking the application status locally through sink interface or remotely via a web interface might need to specify his/her own requirements regarding adaptation.

– *Component-based reconfiguration*: updates in WiSeKit can take place both at component attribute level and at component level. WiSeKit expects application developers implement predefined interfaces for components which are subject to reconfiguration. We present later in this section the signature of such interfaces and the mechanisms for reconfiguration.

– *Distribution*: The heterogeneity of WSNs in terms of the node's resource capabilities and functionalities necessitates support for distribution at the middleware layer in order to achieve the above goals and also optimize network resources usage. WiSeKit is within all nodes types built up over a set of *Core Services* which provides an infrastructure for distribution.

Figure 8.2 illustrates the complete logical architecture of the WiSeKit middleware distributed over the different node types. It shows how the adaptation services are located in different node types and mapped to a typical WSN architecture. At the left side of the figure, sensor node features a set of services for realizing *Local-adaptivity* and *Component-based reconfiguration*. Next to the sensor nodes, the cluster head has the responsibility of *Intermediate-observation* to observe data and analyze it in terms of adaptation required within the scope of a cluster. Finally, at the right side of Figure 8.2, WiSeKit in the sink node is able to retain the "whole" WSN application in a high degree of adaptivity via *Intermediate-observation* and *Remote-observation*. Therefore, the end user of the application can specify his/her own adaptation needs through the APIs provided within the sink node. Middleware services in different nodes interact through core services customized for each type of node. The details of WiSeKit services within each type of node are explained in the rest of this section.

8.4.1 Sensor Side

To address the middleware requirements of adaptive applications, we need first to explore the desired structure of an application deployed over the sensor nodes, then the adaptation middleware services will be discovered accordingly. Figure 8.3 illustrates a sample configuration of application components for a home monitoring application. There are three main aspects that should be considered for application development.

Firstly, the components which are subject to reconfiguration should implement the relevant interfaces. In general, four types of reconfigurations are likely to happen during the application runtime, including: *i)* replacing a component with a new one, *ii)* adding a

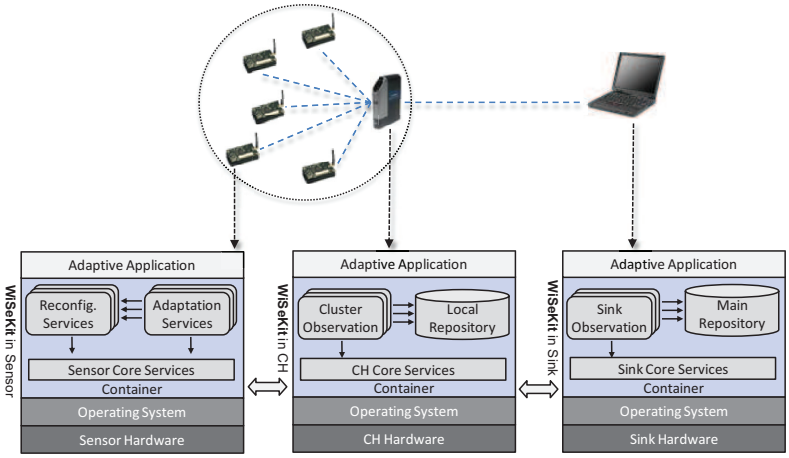

Figure 8.2: WiSeKit in the hierarchical WSN architecture.

new component, *iii)* component removal, and *iv)* changing the values of component member variables. For each type of reconfiguration the relevant interface(s) should be implemented. We explain later in this section the name and specification of those interfaces.

Secondly, as shown in the figure, the deployable package should include a predefined *policy file* describing situation-actions rules. It is one of the main sources of *local* adaptation decision. The local decision is limited to changing the values of component member variables, while the decision of full component image replacement is made by the cluster head. It is because the decisions for replacing or adding components fall in the category of heavyweight reconfiguration requests. Such a decision should be assigned to a node being aware of what happens in the scope of a cluster. Moreover, sometimes we need to send a component image to a set of sensor nodes having similar attributes or responsibilities.

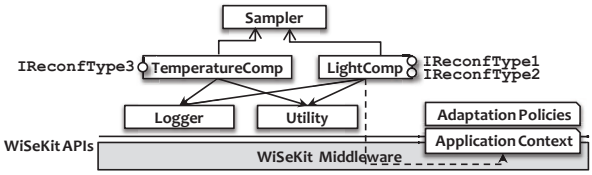

Figure 8.3: A sample component configuration for an adaptive home application.

Finally, *Application Context* is a meta-data reporting application-related contextual

information. Application components can update it when a contextual event is detected through APIs provided by WiSeKit. The content of application context is used together with sensor context information against the situations described in the policy object to check whether any adaption is needed.

Figure 8.4 describes the architecture of our adaptation middleware for sensor nodes. As shown, WiSeKit addresses three main areas of adaptation concern.

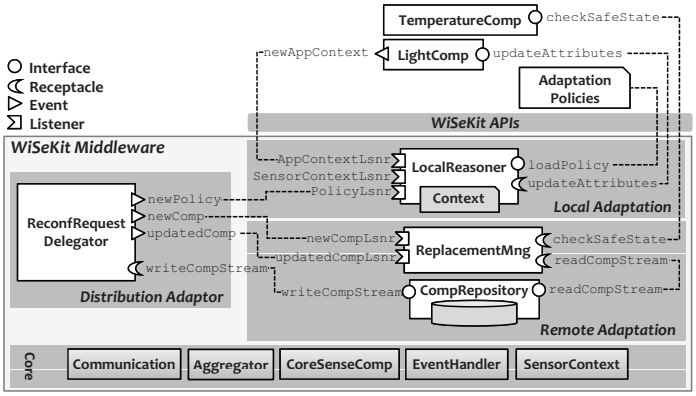

Figure 8.4: WiSeKit services in the sensor node.

Local Adaptation is in charge of carrying out local parameter adaptation requests. LocalReasoner, as the main service receives both the adaptation policies of the application and context information, then it checks periodically the situations within policy file against application context and sensor context for adaptation reasoning. Upon satisfying a situation, the corresponding action, changing the values(s) of component attribute(s), is performed via calling the `updateAttributes` interface of the component and passing the new values.

Remote Adaptation is concerned with adapting the whole component. In fact, the corresponding cluster node performs the reasoning task and notifies the sensor node the result containing which component should be wholly reconfigured. The key service in this part is ReplacementMng. Upon receiving either `newComp` or `updatedComp` event, it reads the component image from CompRepository, loads the new component and finally removes the image stored by CompRepository from the repository.

After loading the component image, the current component is periodically checked to identify whether it is in a *safe state of reconfiguration* or not. The safe state is defined as a situation in which all instances of a component are temporarily idle or not processing any request. There are several safe state checking mechanisms in the literature [19], [20]. In some solutions, safe state recognition is the responsibility of the underlying reconfiguration service, whereas in the other mechanisms this checking is assigned to the component itself.

We adopt the second method because of its low overhead. Therefore, WiSeKit expects from each reconfigurable component to implement the `checkSafeState` interface.

Distribution Adaptor provides a distribution platform for adaptation decision and accomplishment. Specifically, it is proposed to address three issues: *i)* the possibility of updating adaptation policies during application lifespan, *ii)* receiving the result of high-level adaptation decision from cluster head, *i.e.*, the image of a component, and *iii)* providing an abstraction for distributed interactions between WiSeKit services. ReconfRequestDelegator reads the data received through the Communication service, checks whether it encompasses any event such as new policy, new component, or updated component, and finally unmarshals the content and generates the corresponding event object.

The bottom part of middleware is decorated with the *core* to provide an infrastructure for distribution as well as the utility and common services. The Communication service has the responsibility of establishing connection to the other nodes in the hierarchical structure. This service not only sends the data, but also receives the reconfiguration information (component image or policy). Aggregator is a service for performing aggregation of data received from CoreSenseComp. EventHandler handles events generated by the services within the middleware. The context information related to the sensor hardware and system software is reported by SensorContext service as an `newSensorContext` event.

8.4.2 Cluster Head Side

Based on *hierarchical adaptation*, when the context information of a sensor node is not sufficient to make an adaptation decision, the cluster head attempts to decide on an adaptation based on the data received from sensor nodes in its own cluster. Similarly, if the current information in the cluster head is not enough for the adaptation reasoning, the final decision is left to the sink node, *e.g.*, in our motivation scenario, if the occupancy sensors detect a movement in the living room, the cluster head notifies the temperature sensors to reduce the sampling rate. Figure 8.5 illustrates both the structure of WSN application and the WiSeKit architecture over the cluster head. The WiSeKit services within the cluster head make the high-level adaptation decisions through processing *application context model* and *cluster-level adaptation policies*.

The context model defines the possible contextual situations by describing the relations between the application data gathered by sensor nodes [17]. For example, "room occupied" is a situation that can be deduced from checking the data values of both occupancy sensors and light sensors in a room. The cluster-level adaptation policies are described in the same way as for sensor nodes (situation-action). However, in this case, the situations are those defined in the context model. The action also includes loading a new policy or a new component in some selected nodes.

As depicted in Figure 8.5, WiSeKit aims at addressing the high-level reasoning issues within the cluster head. To this end, the middleware services expect from the application to provide: *i)* the context model, and *ii)* the adaptation policies. In this way, WiSeKit

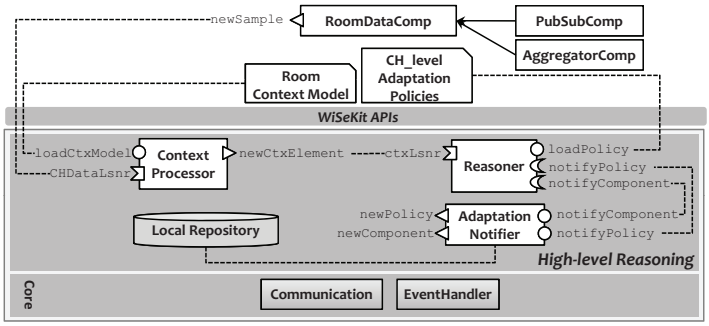

Figure 8.5: WiSeKit in the cluster head.

processes at first the context model along with the data received from sensor nodes in order to find out the current context situation(s) of environment, then the Reasoner service checks whether any adaption is needed. In fact, this service analyzes the adaptation policies based on the current context information, thereby it decides on update notifications, *i.e.*, either new policy or new component. If Reasoner makes the decision of a component update, AdptationNotifier loads the binary object of new component from the *Local Repository* and multicasts it along with the required meta-information to the nodes in its vicinity. AdaptationNotifier is also responsible for forwarding the adaptation requests of the sink node to the cluster members. We assume that the local repository of cluster head contains all versions of a component that might be needed during the application lifespan.

8.4.3 Sink Side

WiSeKit in the sink node is designed in the same way as it is proposed for the cluster head. The main differences between the sink node and the cluster head in the context of our middleware are in two aspects. Firstly, the scope of network covered by the sink node is the whole sensor network, while the cluster head has only access to the information retrieved within a cluster. Therefore, the global adaptation, the highest level of adaptation decision, takes place only in the sink node, where it has access to the status of all clusters. Secondly, the sink node is able to receive end-user preferences regarding to the adaptation requirements.

The component repository within the sink node contains all versions of all components. As the sink node is a powerful node with sufficient resources for processing tasks and storing application components, WiSeKit in the sink node has the ability of reasoning on the sophisticated adaptations and providing different versions of a component according to the adaptation needs.

The communication service within the core of sink provides the following functional-

ities: *i)* global context information exchange between the sink and the cluster heads, *ii)* code distribution, and *iii)* internetworking between WSNs and external networks, *e.g.*, the Internet. While the context information can be piggybacked into either the code distribution or routing protocols to reduce the signaling overhead, the internetworking provides more flexible facilities to manage and control WSNs remotely.

8.5 Preliminary Evaluation

As our adaptation middleware is customized for each type of node, the evaluation should take into account many performance figures. At first, we need to evaluate each type of node separately, then the effectiveness of WiSeKit should be assessed for all nodes together. As considering the evaluation for all nodes is a huge work, this paper focuses only on middleware performance in the sensor node as the critical part of our proposal, while evaluating the whole adaptation middleware is a part of our future work.

The efficiency of our approach for sensor node can be considered from the following performance figures:

- The *memory overhead* of middleware run-time, with respect to both program and data memory. The former can be evaluated by measuring the size of binary images after compilation. The latter includes the data structures used by the programs.
- The *energy usage* during adaptation, which refers to the energy overhead of running an adaptation task.
- The *communication overhead* between sensor nodes and cluster head in the presence of middleware for a typical adaptive application.

We chose the Instant Contiki simulator [21] to measure the overhead of memory. The prototype implementation shows the memory footprint for reconfiguration program and its data is no more than 3 Kbytes in total. As most of sensor nodes are equipped with more than 48 Kbytes of program flash (TelosB node), WiSeKit does not impose a high overhead in terms of memory. It should be noted that this cost is paid once and for all, regardless of the amount of memory is needed for the application components. There is also an application level memory overhead for the description of adaptation policies and implementing the reconfiguration interfaces (`checkSafeState`, `updateAttributes`, etc.). This cost depends directly on the degree of application adaptivity. Moreover, the amount of memory used by CompRepository varies with respect to the number of new components downloaded simultaneously in the sensor node. As WiSeKit removes the image of a component from repository when loading it to the memory, this overhead is kept at a very low level in the order of zero.

For measuring energy consumption, we assume that our hypothetical WSN application is similar to the configuration depicted in Figure 8.6 and Sampler is the replacement candidate. The main reconfiguration tasks include: *i)* checking the old Sampler to ensure that it

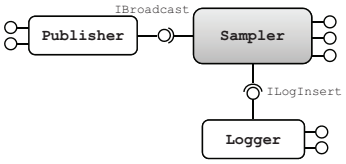

Figure 8.6: Sample configuration.

is not in interaction with the other components before starting reconfiguration, *ii)* saving the state of old Sampler, *iii)* creating the new one and transferring the last state to it.

Each loadable module in Contiki is in *Compact Executable and Linkable Format* (CELF) containing code, data, and reference to other functions and variable of system. When a CELF file is loaded to a node, the dynamic linker in core resolves all external and internal references, and then writes code to ROM and the data to RAM [22]. For our sample configuration, the Sampler_CELF file (764 bytes) must be transferred to the node, and all mentioned tasks for dynamic loading must be done for the Sampler program (its code size is 348 bytes). As the energy consumption depends on the size of new update, the model of energy consumption will be [22]:

$$E = S_{New\_CELF} \times (P_p + P_s + P_l) + S_{New\_Sampler} \times P_f + E_{safeStateCheck}$$

Where S_{New_CELF} is the size of new CELF file and P_p, P_s, P_l and P_f are scale factors for network protocol, storing binary, linking and loading, respectively. $S_{New\_Sampler}$ is the code size of new Sampler, and $E_{safeStateCheck}$ is the energy cost of performing reconfiguration. Concretely, we obtain the following energy consumption for the considered adaptation scenario:

$$E = 764 \times (P_p + P_s + P_l) + 348 \times P_f + E_{safeStateCheck}$$

In this equation, we take into account the overhead of checking safe state (dependencies to the other two components). We believe that this value is very low compared to the first part, which is the reconfiguration cost imposed by Contiki.

To measure the communication overhead, we assume a scenario in the living room of home application in which the "occupancy" of context changes occasionally. According to the monitoring rules of home, when the room is empty the temperature sensors should report the temperature of room every 10 minutes. Once the room is occupied the temperature sensors should stop sensing until the room becomes empty again.

According to this scenario, when the room is occupied, ContextProcessor within the cluster head observes the new context and Reasoner notifies the relevant sensor nodes to stop sampling. WiSeKit does not impose any communication cost for context detection because it piggybacks the current value of a node's attributes at middleware layer of cluster

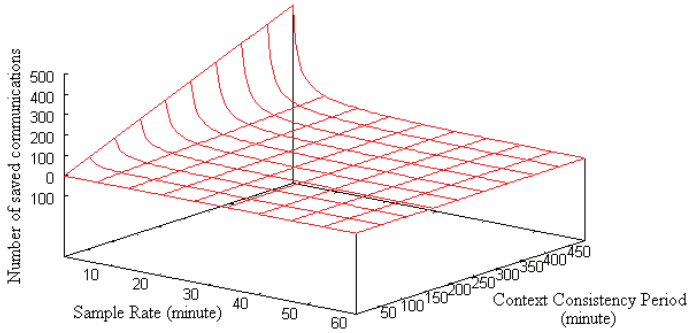

Figure 8.7: Number of saved communications for a sample home monitoring scenario.

head. Therefore, the communication cost is limited to sending policy objects to stop/restart sampling task.

Three parameters should be taken into account to measure the overhead of communication: *i)* sampling rate (r), *ii)* context consistency duration (c), and *iii)* number of context changes during a particular period of time (k). For the hypothesis scenario, if a room is occupied for two hours during one day, we have: $r = 10$ min, $c = 120$ min, and $k = 1$. In this case, the temperature sensors do not send the data for two hours, thus the number of communication for one day (24 hours) is:

$$\begin{aligned} N_{total} &= N_{forWholeDay} - N_{occupiedTime} + N_{WiSeKitOverhead} \\ &= (24 \times 60)/r - c/r + k \times N_{policySending} \\ &= 144 - 12 + 2 = 134 \end{aligned}$$

Therefore for this case the number of saved communications is 10. Generally, we can evaluate that the saved number of communications is:

$$\begin{cases} N_{saved} = c/r \times k - 2 \times k \\ 1 \leqslant k \leqslant 24/y \end{cases}$$

Figure 8.7 shows the saved number of communication. As the consistency period of new context is increased and sampling rate is decreased, more number of communications will be saved. This is because the middleware prevents a sensor node to send data during the period of new context activation.

8.6 Related Work

The first prominent work reported to address reconfigurability for resource-constrained systems is [6]. In this paper, Costa et al. propose a middleware framework for embedded

systems. Their approach focuses on a kernel providing primary services needed in a typical resource-limited node. Specifically, their work supports customizable component-based middleware services that can be tailored for particular embedded systems. In other words, this approach enables reconfigurability at the middleware level, while our proposal tries to give this ability to the application services through underlying middleware services.

Efforts for achieving adaptivity in WSNs have continued by Horr et al [5]. They proposed DAVIM, an adaptable middleware enabling dynamic service management and application isolation. Particularly, their main focus in this work is on the composition of reusable services in order to meet the requirements of simultaneously running applications. In fact, they consider the adaptivity from the view of dynamic integration of services, whereas our work tries to make the services adaptable.

A fractal composition-based approach for constructing and dynamically reconfiguring WSN applications is introduced in [23]. The approach uses π-calculus semantics to unify the models of interaction for both software and hardware components. The novel feature of that approach is its support for a uniform model of interaction between all components, namely communication via typed channels. Although the reconfiguration model in [23] is promising, it fails to explain under which conditions a reconfiguration should take place.

The most relevant work in the context of reconfiguration for WSN has been reported recently under the name of FiGaRo framework [8]. The main contribution of FiGaRo is to present an approach for *what and where* should be reconfigured. The former one is related to runtime component replacement, and latter is concern with which nodes in the network should receive update code. In fact, FiGaRo provides a run-time support for component loading, while our approach proposes a generic solution which includes all of-the-shelf adaptation services besides the feature of run-time component loading.

8.7 Conclusions and Future Work

In this paper, we proposed WiSeKit as a middleware solution making adaptation and reconfiguration of WSN application software possible. We categorized our proposal into three different layers according to the hierarchical architecture of WSN and presented WiSeKit features for each type of node. The hierarchical adaptation decision of WiSeKit conforms the hierarchical architecture of WSNs so that based on the resource availability in a node as well as the portion of the network covered by a node, adaptation and reconfiguration are performed.

This paper focused only on adaptation for the portion of application running on sensor nodes, while the part of application deployed on cluster head and sink may need to be adapted as well. This issue will be addressed in our future work. The work reported in this paper is a part of our comprehensive solution for self management in WSNs. Integrating this work with the other work reported in [9], [17] is another future direction. Developing a complete home monitoring application based on the proposed middleware is also included in the plan for future work.

References

[1] D. Puccinelli and M. Haenggi, "Wireless sensor networks: applications and challenges of ubiquitous sensing," *Circuits and Systems Mag., IEEE*, vol. 5, no. 3, pp. 19–31, 2005.

[2] P. e. a. Costa, "The runes middleware for networked embedded systems and its application in a disaster management scenario," in *PERCOM '07: Proc. of the Fifth IEEE Int. Conf. on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2007, pp. 69–78.

[3] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks*, vol. 2, no. 4, pp. 351–367, 2004.

[4] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proc. of the 29th Annual IEEE Int. Conf. on Local Computer Networks*. Tampa, Florida, USA: IEEE Computer Society, 2004, pp. 455–462.

[5] W. Horré, S. Michiels, W. Joosen, and P. Verbaeten, "Davim: Adaptable middleware for sensor networks," *IEEE Distributed Systems Online*, vol. 9, no. 1, p. 1, 2008.

[6] P. Costa, G. Coulson, C. Mascolo, L. Mottola, G. P. Picco, and S. Zachariadis, "A reconfigurable component-based middleware for networked embedded systems," *Journal of Wireless Information Networks*, vol. 14, no. 2, 2007.

[7] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes, "Dynamic reconfiguration in sensor middleware," in *MidSens '06: Proc. of the Int. Workshop on Middleware for sensor networks*. Melbourne, Australia: ACM, 2006, pp. 1–6.

[8] L. Mottola, G. P. Picco, and A. A. Sheikh, "Figaro: fine-grained software reconfiguration for wireless sensor networks," in *EWSN '08: Proc. of the 5th European Conf. on WSNs*. Bologna, Italy: Springer-Verlag, 2008, pp. 286–304.

[9] A. Taherkordi, F. Eliassen, R. Rouvoy, and Q. Le-Trung, "Rewise: A new component model for lightweight software reconfiguration in wireless sensor networks," in *IWSSA '8: Proc. of the 7th Int. Workshop On System/Software Architectures*. Monterrey, Mexico: Springer-Verlag, 2008, pp. 415–425.

[10] M. C. Huebscher and J. A. McCann, "Adaptive middleware for context-aware applications in smart-homes," in *MPAC '04: Proc. of the 2nd workshop on Middleware for pervasive and ad-hoc computing*. Toronto, Canada: ACM, 2004, pp. 111–116.

[11] M. C. Mozer, "Lessons from an adaptive home," *Smart Environments: Technology, Protocols, and Applications*, pp. 273–298, 2004.

[12] V. Poladian, J. P. Sousa, D. Garlan, and M. Shaw, "Dynamic configuration of resource-aware services," in *ICSE '04: Proc. of the 26th Int. Conf. on Software Engineering*. Washington, DC, USA: IEEE Computer Society, 2004, pp. 604–613.

[13] D. Garlan, S.-W. Cheng, A.-C. Huang, B. Schmerl, and P. Steenkiste, "Rainbow: Architecture-based self-adaptation with reusable infrastructure," *Computer*, vol. 37, pp. 46–54, 2004.

[14] H. Lutfiyya, G. Molenkamp, M. Katchabaw, and M. A. Bauer, "Issues in managing soft qos requirements in distributed systems using a policy-based framework," in *POLICY '01: Proc. of the Int. Workshop on Policies for Distributed Systems and Networks*. London, UK: Springer-Verlag, 2001, pp. 185–201.

[15] J. O. Kephart and D. M. Chess, "The vision of autonomic computing," *Computer*, vol. 36, pp. 41–50, 2003.

[16] J. O. Kephart and R. Das, "Achieving self-management via utility functions," *IEEE Internet Computing*, vol. 11, pp. 40–48, 2007.

[17] A. Taherkordi, R. Rouvoy, Q. Le-Trung, and F. Eliassen, "A self-adaptive context processing framework for wireless sensor networks," in *MidSens '08: Proc. of the 3rd Int. Workshop on Middleware for WSNs*. Leuven, Belgium: ACM, 2008, pp. 7–12.

[18] Q. Le-Trung, A. Taherkordi, T. Skeie, H. N. Pham, and P. E. Engelstad, "Information storage, reduction and dissemination in sensor networks: a survey," in *CCNC '09: Proc. of the 6th IEEE Conf. on Consumer Communications and Networking*. Las Vegas, NV, USA: IEEE Press, 2009, pp. 1111–1116.

[19] J. a. P. A. Almeida, M. Van Sinderen, and L. Nieuwenhuis, "Transparent dynamic reconfiguration for CORBA," in *DOA '01: Proc. of the 3rd Int. Symp. on Dist. Objects and Apps*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 197–.

[20] J. Zhang, B. H. Cheng, Z. Yang, and P. K. McKinley, "Enabling safe dynamic component-based software adaptation," in *Architecting Dependable Systems III*, ser. Lecture Notes in Computer Science. Springer, 2005, vol. 3549, pp. 194–211.

[21] InstantContiki, 2008, http://www.sics.se/contiki.

[22] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, "Run-time dynamic linking for reprogramming wireless sensor networks," in *SenSys '06: Proc. of the 4th Int. Conf. on Embedded networked sensor systems*. Colorado, USA: ACM, 2006, pp. 15–28.

[23] D. Balasubramaniam, A. Dearle, and R. Morrison, "A composition-based approach to the construction and dynamic reconfiguration of wireless sensor network applications," in *SC '08: Proc. of the 7th Int. Conf. on Software composition*. Budapest, Hungary: Springer-Verlag, 2008, pp. 206–214.

Chapter 9

A Generic Component-based Approach for Programming, Composing and Tuning Sensor Software

Authors. Amirhosein Taherkordi (1), Frédéric Loiret (2), Romain Rouvoy (1,2), and Frank Eliassen (1)

Affiliation.

(1) Department of Informatics, University of Oslo, Norway
{amirhost,rouvoy,frank}@ifi.uio.no

(2) INRIA Lille Nord Europe, ADAM Project-team, University Lille 1, LIFL CNRS UMR 8022, Villeneuve dAscq, France
{frederic.loiret,romain.rouvoy}@inria.fr

Publication. The Computer Journal, Oxford University Press. The earlier version of this paper has been published in the proceedings of the 6th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS'10), Springer-Verlag, Santa Barbara, California, USA, June 2010.

Abstract. *Wireless Sensor Networks* (WSNs) are being extensively deployed today in various monitoring and control applications by enabling rapid deployments at low cost and with high flexibility. However, high-level software development is still one of the major challenges to wide-spread WSN adoption. The success of high-level programming

approaches in WSNs is heavily dependent on factors like ease of programming, code well-structuring, degree of code reusability, required software development effort, and the ability to tune the sensor software for a particular application. Component-based programming has been recognized as an effective approach to satisfy such requirements. However, most of the componentization efforts in WSNs were ineffective due to various reasons, such as high resource demand or limited scope of use. In this article, we present REMORA, a novel component-based approach to overcome the hurdles of WSN software implementation and configuration. REMORA offers a well-structured programming paradigm that fits very well with resource limitations of embedded systems, including WSNs. Furthermore, the special attention to event handling in REMORA makes our proposal more practical for embedded applications, which are inherently event-driven. More importantly, the mutualism between REMORA and underlying system software promises a new direction towards separation of concerns in WSNs. This feature also offers a practical way to develop sensor *middleware services* which should be generic and developed close to the operating system. Additionally, it allows the customization of sensor software—deploying only application-required system-level services on nodes, instead of installing a fixed large system software image for any application. Our evaluation results show that the deployed REMORA applications have an acceptable memory overhead and a negligible CPU cost compared to the state-of-the-art development models.

9.1 Introduction

Wireless Sensor Networks (WSNs) are a rapidly emerging research area because of their vast application vistas in real-world environments. The advances in wireless communications and miniaturization of hardware components have enabled the development of low-cost, low-power, and multifunctional sensor nodes. These tiny devices can be easily embedded in the environment, establish a wireless ad-hoc network, and compose a distributed system to collaboratively sense and process the surrounding physical phenomenons as data. However, WSNs differ from the conventional distributed systems in many aspects. Resource scarceness is the most important uniqueness of WSNs. Sensor nodes are often equipped with a limited energy source and a processing unit with a small memory capacity. Additionally, the network bandwidth is much lower than for wired communications and radio-based operations are the dominant energy consumer within a sensor node. The sensor nodes and network are less reliable than in conventional distributed systems. Depending upon the configuration of network and environment circumstances, wireless links may become degraded or unviable.

These factors make the way to develop WSN applications quite critical and also different from the other existing network systems. However, this concept is still immature in the context of WSNs for various reasons. Firstly, the existing diversities in WSN hardware and software platforms have brought the same order of diversity to programming models for such platforms [1]. Moreover, developers' expertise in state-of-the-art programming

models become useless in WSN programming as the well-established discipline of program specification is largely missing in this area. Secondly, the structure of programming models for WSNs are usually sacrificed for resource usage efficiency, thereby, the outcome of such models is usually a piece of tangled code hardly maintainable by its owner. Finally, application programming in WSNs is mostly carried out very close to the operating system, forcing developers to learn low-level system programming models. This not only diverts the programmer's focus from the application logic, but also needs low-level programming techniques, which imposes a significant burden on the programmer.

From a software composition perspective, the way to implement WSN applications is also becoming increasingly important as today's sensor software not only consists of application and system modules, but also includes various off-the-shelf, third-party software products, such as middleware services. Ideally, such integrations should be realized through a meta-level abstraction with minimum programming effort. This, in fact, indicates the capability of a WSN programming model to facilitate the development of middleware services and their integration to target application software.

The ability to tune the sensor software for a particular use-case or application domain is the other major issue in this context. As sensor nodes are typically equipped with a limited memory capacity, operating system developers need to keep the size of system modules as small as possible in order to preserve enough memory space for application modules, and they also have to ensure the portability of system software to various sensor platforms. This mostly leads to software artifacts with either degraded functionality not satisfying all end-user expectations, or suffering from the lack of modularity and maintainability. One solution to tackle this problem is to consider the operating system as a collection of well-defined services deployable on a minimized kernel image so that the programmer has the ability to involve only application-required system services in the process of software installation. Therefore, this can bring a significant efficiency to resource usage in sensor nodes by avoiding installing a single monolithic operating system for any application.

Software *componentization* has been recognized as a well-structured programming model able to tackle the above concerns. Component-based programming provides an high-level programming abstraction by enforcing interface-based interactions between system modules and therefore avoiding any hidden interaction via direct function call, variable access, or inheritance relationships. This abstraction rather offers the capability of black-box integration of modules in order to simplify configuration and maintenance of software systems. Module reusability and provision of standard API are some other advantages of adopting *component-based software development* [2, 3]. Although using this paradigm in earlier embedded systems was relatively successful [4, 5, 6, 7], most of the efforts in the context of WSNs remain inefficient or limited in the scope of use. The TinyOS programming model, named NesC [8], is perhaps the most popular component model for WSNs. Whereas NesC eases WSN programming, this component model remains tightly bound to the TinyOS platform. Other proposals, such as OpenCom [9] and THINK [10], are either too heavyweight for WSNs, or not able to support event-driven programming, which is of

high importance in WSNs.

In this article, we present extended results on REMORA, a lightweight component model designed for resource-constraint embedded systems, including WSNs [11]. The strong abstraction promoted by this model allows a wide range of embedded systems to exploit it at different software levels from *Operating System* (OS) to application. To achieve this goal, REMORA provides a very efficient mechanism for event management, as embedded applications are inherently event-driven. REMORA components are described in XML as an extension of the *Service Component Architecture* (SCA) model [12] in order to make WSN applications compliant with the state-of-the-art componentization standards. Additionally, the C-like language for component implementation in REMORA attracts both embedded system programmers and PC-based developers to programming for WSNs. REMORA also features a coherent mechanism for component *instantiation* and *property-based component configuration* in order to facilitate lightweight event-driven programming in WSNs. Notably, in this paper the aforementioned features of REMORA are extended in the following ways. First, we propose a programming approach, based on the concept of *Autonomous Composable Module* (ACM), to achieve a practical and efficient way of developing component-based *middleware systems* in WSNs. Second, we introduce a mechanism to enable *tuning* system software by componentizing the OS-level services and customizing OS functionality based on target application's requirements. The REMORA specifications and their implementation techniques are also extensively explored in this paper.

As a matter of validation, we demonstrate the comprehensive evaluation results of deploying REMORA components on Contiki—a leading operating system for WSNs [13]. Specifically, we extend our earlier evaluation efforts in [11] with considering a complementary set of performance figures, such as required programming effort. The efficient use of Contiki features, such as process management and event distribution [14], on the one hand, and the abstraction layer linking REMORA to Contiki, on the other hand, promise a very effective and generic approach towards practical high-level programming in WSNs. In particular, we present the functionality of REMORA within the context of a real use case involving a network-level *application suite* in order to support *code distribution* in dynamic sensor applications. Finally, the evaluation work is completed by carrying out a comprehensive investigation of existing software component models for WSNs and comparing them with REMORA.

The remainder of this article is therefore organized as follows. In Section 9.2, the specification of the REMORA component model is presented. Section 9.3 describes how REMORA is implemented, while the evaluation results are reported in Section 9.4, including the assessment of a real REMORA-based deployment. A survey of existing approaches and a discussion on REMORA extension opportunities are presented in Section 9.5 and Section 9.6, respectively. Finally, Section 9.7 concludes this paper and identifies some future work.

9.2 Remora Component Model

In this section, we first discuss the primary design concepts in Remora and then we explain the specifications of the Remora component model. The first obvious principle is that WSN applications in our approach are built out of components conforming to the Remora component model. The other design principles of Remora include:

XML-based Component Description. The first design goal emphasizes simplicity and generality of the technique for describing Remora components. In Remora, we therefore adopt XML technologies to describe components. The basis for the XML schema we defined is the *Service Component Architecture* (SCA) notations in order to provide a uniform component model covering components from sensors to the Internet, as well as to accelerate standardization of component-based programming in WSNs. As SCA was originally designed for large-scale systems-of-systems [12], Remora extends SCA with its own architectural concerns to achieve realistic component-based programming in WSNs.

C-like Language for Component Implementation. Remora components are written in a C-like language enhancing the C language with features to support component-based and structured programming. The other objective in this enhancement is to attract both embedded systems programmers and PC-based developers towards high-level programming in WSNs.

OS Abstraction Layer. The Remora component framework is integrated with the underlying operating system through a well-defined OS-abstraction layer. This thin layer can be developed for various WSN operating systems supporting the C language, such as Contiki. This feature ensures the portability of Remora components towards different OSs. The abstraction provided by Remora becomes more valuable when the component framework is easily configured to reuse OS-provided features, such as event processing and task scheduling.

Event Handling. Event-driven programming is a common technique for programming embedded systems as memory requirements in this programming model is very low. Besides the support for events at the operating system level in embedded systems, we also need to consider event handling at the application layer. Remora therefore proposes an high-level support of event generation and event handling, which makes it one of the key features of our proposal. In particular, Remora achieves this goal by reifying the concept of event as a first-class architectural element simplifying the development of event-oriented scenarios.

Before describing our component model, we first define the basic terms used throughout this article. Figure 9.1 illustrates the development process of Remora-based applications. A Remora application consists of a set of Remora Components, containing descriptions and implementations of software modules. The Remora *engine* processes the components and generates standard C code deployable within the Remora *framework*. The framework is an OS-independent module supporting the specification of the Remora component model. Finally, the Remora application is deployed on the target sensor node via the Remora *runtime*, which is an OS-abstraction layer integrating the application to the

system software.

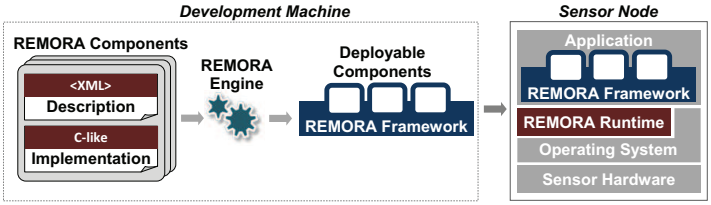

Figure 9.1: Development process of REMORA-based applications.

9.2.1 Component Specification

A REMORA component contains two main artifacts: *component description* and *component implementation*. The component description is an XML document containing the specifications of the component including its *services*, *references*, *producedEvents*, *consumedEvents*, and *properties* (cf. Figure 9.2). A *service* can expose a REMORA interface, which is a separate XML document specifying the functions provided by the component, while a *reference* indicates the operations required by the component as an interface. Likewise, a *producedEvent* identifies an event type generated by a component, whereas a *consumedEvent* specifies component's interest on receiving a particular event. The component implementation is a C-like program containing three types of operations: *i)* operations implementing the component's services, *ii)* operations processing events, and *iii)* component's private operations.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="COMPONENT_NAME">
  <service name="SERVICE1_NAME">
    <interface.remora name="INTERFACE1_NAME"/>
  </service>
  ... other services
  <reference name="REFERENCE1_NAME">
    <interface.remora name="INTERFACE2_NAME"/>
  </reference>
  ... other references
  <property name="PROP1_NAME" type="PROP1_TYPE">
    PROP1_DEFAULT_VALUE
  </property>
  ... other properties
  <producer>
    <event.remora type="EVENT1_TYPE" name="EVENT1_VAR_NAME"/>
  </producer>
  ... other producers
  <consumer operation="CONSUMER_OPERATION">
    <event.remora type="EVENT2_TYPE" name="EVENT2_VAR_NAME"/>
  </consumer>
  ... other consumers
</componentType>
```

Figure 9.2: The XML template for describing REMORA components.

To make the specification more concrete, we first present a simple example of a REMORA-based application, then we discuss REMORA features carefully. This simple application is in charge of *blinking* a LED on a sensor node every three seconds. Figure 9.3 depicts the components involved in this application.

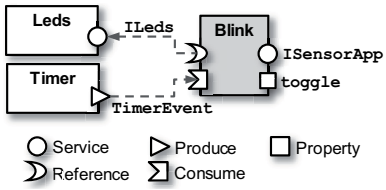

Figure 9.3: A simple REMORA-based application.

We here focus on the Blink component and describe it according to the REMORA component model. In Figure 9.4, the XML description of the Blink component is shown. This component provides an `ISensorApp` interface to start application execution and requires an `ILeds` interface to switch LEDs on and off, which is implemented by the Leds component. It also owns a property to `toggle` a LED on the sensor node. As the Blink component produces no event, the `producer` tag in the component description is empty, while it is subscribed to receive TimerEvent and process this event in the `timerExpired` function. The last part of the component description is the libraries used by the component implementation.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="app.BlinkApp">
  <service name="iSensorApp">
    <interface.remora name="core.boot.api.ISensorApp"/>
  </service>
  <reference name="iLeds">
    <interface.remora name="core.peripheral.api.ILeds">
  </reference>
  <property name="toggle" type="xsd:short">0</property>
  <producer/>
  <consumer operation="timerExpired">
    <event.remora type="core.sys.TimerEvent" name="aTimeEvent"/>
  </consumer>
  <libraries>
    <include name="stdio" type="SystemLib"/>
  </libraries>
</componentType>
```

Figure 9.4: XML description of Blink component.

Figure 9.5 presents the excerpt of the Blink implementation. This C-like code implements the single function of the `ISensorApp` interface (`runApplication`) and handles TimerEvent within the `timerExpired` function. In the `runApplication` function, we specify that the TimerEvent generator (`aTimeEvent.producer`) is configured to generate periodi-

cally TimeEvent every three seconds. The last command in this function is also to notify the TimerEvent generator to start time measurement. When time is expired, Timer sets the attributes of aTimeEvent (*e.g.*, latency) and then the REMORA framework calls the `timerExpired` function.

```
void runApplication(){
  printf("--- Starting Blink Application ---");
  short periodic = 1;
  aTimeEvent.producer.configure(3*CLOCK_SECOND, periodic);
  aTimeEvent.producer.start();
}
void timerExpired(){
  if (this.toggle == 0){
    iLeds.onLeds(LEDS_RED);
    this.toggle = 1;
  }else{
    iLeds.offLeds(LEDS_RED);
    this.toggle = 0;
  }
  printf("Time elapsed after interval: %d", aTimeEvent.latency);
}
```

Figure 9.5: C-like implementation of Blink component.

Services and References. The first step towards component-based programming is identifying system services, and then identifying which component(s) provides a service and which one(s) requires the service (so called reference). Similar to component descriptions in REMORA, interfaces are described in XML. Interface description includes a name and the associated operations. Figure 9.6 presents the simplified `ILeds` interface used by the Blink component as a reference. Every component providing a service should implement all the operations specified in the interface description with the same signatures.

```
<?xml version="1.0" encoding="UTF-8"?>
<interface.remora name="core.peripheral.api.ILeds">
  <operation name="getLeds" return="xsd:unsignedByte"/>
  <operation name="onLeds">
    <in name="leds" type="xsd:unsignedByte"/>
  </operation>
  <operation name="offLeds">
    <in name="leds" type="xsd:unsignedByte"/>
  </operation>
</interface.remora>
```

Figure 9.6: A simplified description of `ILeds` interface.

Component Properties. In REMORA, programmers can define properties for a component. Properties enable reconfiguration of component behaviors and also convert components from a dead unit of functionality to an active entity tractable during the application lifespan. The component reconfiguration becomes very essential for event producer components, *e.g.*, to generate accurate TimerEvents in the Blink application, we need to configure

the Timer component through a property that holds the time at which the measurement is started. Properties also enable components to become either *stateless* or *stateful*. A component is stateful if and only if it defines a property—*e.g.*, the Blink component in our sample application is a stateful component retaining the value of the `toggle` property—whereas the Leds component is a stateless component. The properties of a component can be accessed from the component implementation using the keyword `this`.

Component Implementation. Remora components are implemented by using a dialect of C language with a set of new commands. This C-like language is mainly proposed to support the unique characteristics of Remora, namely, component instantiation, event processing, and property manipulation. Therefore, for pure component-based programming without the above features, the programmer can almost rely on C features and develop an elementary Remora-based application including only Remora-based interface invocations. We implicitly introduced a few of these commands within the Blink component implementation, while the complete description of commands is available in [15].

Parameter-based Reconfiguration. To preserve efficiency in resource usage, Remora relies on compile-time linking so that system components are linked together statically and their memory address is also computed at compile-time. Additionally, for multiple-instance components, all required instances are created in compiler-specified addresses prior to application startup. These constraints not only reduce the size of the final code, but also relieve the programmer from the burden of managing memory within the source code. In Remora, the reconfiguration feature is also considered from a parametric perspective: A Remora component can be reconfigured statically by changing the behavior of its functions through its component *properties*. In particular, for the property-dependent functions of a component, the behavior of the component can easily be changed by adjusting property values and thus a form of parameter-based reconfigurability is enabled within the component.

9.2.2 Component Instantiation

Remora features a concrete mechanism to support component instantiation. This feature is essentially proposed to manage efficiently event producer components. The Remora engine greatly benefits from component instantiation when undertaking linking of one event producer to several consumer components. For example, in the Blink application, the producer (Timer) of TimerEvent should be instantiated per consumer component, while the UserButtonEvent generator is a single-instance component publishing an event to all subscribed components when the user button on a sensor node is pressed.

By component instantiation, we refer to two principles: *i)* the component code is always single-instance, and *ii)* the component *context* is replicated per instance. By component context, we mean the *data structures* required to handle the properties independently from the component's code. By doing that, a Remora component becomes a *reconfigurable and reusable* entity with a strong abstraction, and more importantly the memory overhead is

kept very low by avoiding code duplication.

REMORA proposes three *multiplicity types* for the component's context: *raw-instance* (stateless component), *single-instance*, and *multiple-instances*. The REMORA engine features an algorithm computing the multiplicity type of a component based on three parameters: *i)* whether the component owns any property, *ii)* whether the component is an event producer, and *iii)* the number of components subscribed to a specific event. When the multiplicity type is determined, the REMORA engine statically allocates memory to each component instance.

9.2.3 Event Management

As high-level event processing is a necessary functionality in embedded systems, the REMORA design comprehensively supports events between components. The main goal is to reify the concept of event as a first-class architectural element simplifying the development of event-oriented scenarios at a low cost. The event design principles in REMORA include:

Event Attributes. An event type in our approach can have a set of attributes with specific types. By defining attributes, the event producer can provide the event-specific information to the event consumer, *e.g.*, the `latency` attribute of TimerEvent in the Blink application.

Application Events vs OS Events. Events in our framework are categorized into two classes: *application-events* and *OS-events*. Application-level events are generated by the REMORA framework (like Timer in the Blink application), while the latter are generated by the sensor operating system. In other words, the only difference of these two types is the source of event generation. To process OS-events at the application level, the REMORA runtime features mechanisms to observe OS-events, translate them to corresponding application-level events, and publish them through *OS-event producer* components.

Event Observation Interface. One of the important aspects of event processing is the time period in which events should be observed by the event producer. Obviously, the length of this period varies with the type of events, *e.g.*, the observation period for a TCP/IP event is the whole application lifespan (*automatic* observation), while a Timer event is observed according to the user-configured time (*manual* observation). REMORA therefore proposes the *event observation interface* in order to control the manual observations. This interface includes event control operations, such as `start`, `pause`, `resume`, and `terminate`. If an event type is manually observable, the associated event producer should implement the generic observation interface. By doing that, the event consumer can handle the life cycle of the observation process by calling the operations of this interface without being aware of the associated event producer.

Event Configuration Interface. The specification of an event type in our approach contains a *configuration interface*. Each component producing an event should implement the associated configuration interface. This feature enables the event consumer to configure event generation before starting the event observation process. More importantly, by

introducing such an interface within the event specification, the event producer and the event consumer become completely decoupled, *e.g.*, in the Blink application, TimerEvent generation is configured within the Blink component without being aware of the associated event generator.

Single Event Producer per Event Type. Each event type in our approach is produced by *one and only one* component. Instead of imposing the high overhead of defining event channels and binding event consumers and producers, we ease event-based programming by assuming one-to-one association between event types and event producers. The programmer is also released from identifying such bindings as the REMORA framework becomes responsible to automatically wire producers and consumers. We believe that this assumption does not affect event-related requirements of embedded platforms. In case an event is produced by two different components, the programmer can define a new event type, extended from the original event, for one of the producer components.

Event Casting. Events in our proposal can be either *unicast*, or *multicast*. Unicast is a one-to-one connection between an event producer and an event consumer—*e.g.*, TimerEvent in the Blink application. In contrast to the unicast model, a multicast event may be of interest to more than one component—*e.g.*, a UserButtonEvent may be handled by several components. The REMORA framework distinguishes between these two types in order to improve the efficiency of processing and distributing events. Event distribution should also be considered together with component instantiation. We need to clarify how multiplicity type of components on the one side, and unicast events and multicast events on the other side are related. To this end, we define two invariants:

> Invariant1: *The consumer of a unicast event should be a raw-instance or single-instance component.*
>
> Invariant2: *The producer of a multicast event should be a raw-instance or single-instance component.*

These invariants are mainly proposed to boost the efficiency of event processing in the REMORA framework. We do not support other event communication schemes since it implies to reify at runtime the source and the destination of an event and to maintain complex routing tables within the REMORA framework, which will induce significant overheads in term of memory footprints and execution time. We rather believe that these invariants do not limit event-related logic of embedded applications.

Events Description. Similar to components, events have their own descriptions, which are in accordance to the event specification in REMORA. Figure 9.7 presents a simplified events description document of the Blink application. This document consists of two outer tags: `remora-events` and `os-events`, corresponding to the application level events and the OS events, respectively. For each event type, we can specify its observation model and casting type. The attributes of an event are also described by the attribute tag and the operations of event configuration interface is specified by the `configInterface` tag.

```
<?xml version="1.0" encoding="UTF-8"?>
<events>
  <remora-events>
    <event-type name="core.sys.TimerEvent"
                    castType="unicast" observation="manual">
      <attribute name="latency" type="xsd:int"/>
      <configInterface>
        <operation name="configure">
          <in name="interval" type="xsd:int"/>
          <in name="periodic" type="xsd:short"/>
        </operation>
      </configInterface>
    </event-type>
  </remora-events>
  <!-- add other application event types here -->
  <os-events>
    <!-- describe OS-events here -->
  </os-events>
</events>
```

Figure 9.7: Application events description.

Event Management Illustration. Figure 9.8 illustrates the event management mechanism implemented in REMORA. We explain the mechanism based on the steps labeled in the figure. During the first two steps, the event consumer can configure event generation and control event observation by calling the associated interfaces realized by the event producer component. These steps in our sample application are achieved in the Blink component (event consumer) by the code below:

```
aTimeEvent.producer.configure(3*CLOCK_SECOND, periodic);
aTimeEvent.producer.start();
```

Note that the programmer is not aware of the TimerEvent producer. She/he only knows that the TimerEvent generator is expected to implement the `configure` function defined in the description of TimerEvent (cf. Figure 9.8). The TimerEvent producer should also implement the observation interface as the observation type of TimerEvent is manual.

Whereas the above steps are initiated by the component programmer, the next two steps are performed by the REMORA component framework. Step 3 is dedicated to *polling* the producer component to observe event occurrence. The event producer is polled by the REMORA framework through a *dispatcher* function in the producer. In fact, the event observation occurs in this function. The polling process is started, paused, resumed, and terminated based on the programmer's configuration for the event observation, performed in step 2.

For application-level events, the REMORA framework is in charge of calling periodically this function, while for OS-events, REMORA invokes this function whenever an OS-event is observed by the REMORA runtime. The REMORA runtime listens only to application-requested OS-events, and delivers the relevant ones to the framework. The REMORA framework then forwards the event to the corresponding OS-event producer component by calling its dispatcher function—*e.g.*, `user_button` is a Contiki-level event that should

be processed by the REMORA component UserButton. This component then generates an high-level UserButtonEvent and publishes it to the REMORA framework.

Finally, in step 4, upon detecting an event in the dispatcher function, the producer component creates the associated event, fills the required attributes, and publishes it to the REMORA framework. The framework in turn forwards the event to the interesting components by calling their event handler function.

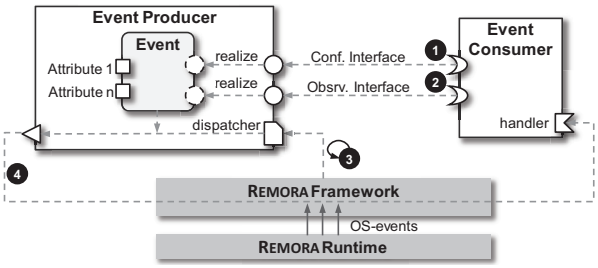

Figure 9.8: Event management mechanism in REMORA.

9.2.4 Components Assembly and Deployment

A typical REMORA application may contain several implementations of a given component type due to the existing heterogeneity in WSN hardware and software platforms. To configure an application according to the target platform requirements, REMORA introduces components *assembly* (equivalent to *composite* component in SCA). This XML document specifies the list of application components, as well as bindings between references and services of components. Figure 9.9 shows the configuration of Blink application in which there is only one binding from Blink to the Leds component implementing the interface `ILeds` for the MSP430 micro-controller. Note that, based on the event casting invariants, the event-binding between Blink and Timer is created automatically by the REMORA framework.

Figure 9.10 illustrates the four main phases of an application deployment. The REMORA development box encompasses artifacts supporting component specification. Events description and components configuration are used to describe system events and components assembly, respectively. Components and interfaces are also described in separate XML documents, one for each. External types are a set of C header files containing application's type definitions. The last group of elements in this box are C-like implementation files of components in which OS libraries may be called through a set of System APIs implemented by REMORA runtime components. Note that there is no hard-coded dependencies between REMORA implementers and the native API of the underlying OS (*e.g.*, Contiki) to ensure portability of REMORA components towards different OSs.

```
<?xml version="1.0" encoding="UTF-8"?>
<composite name="app.BlinkAppConfigurer">
  <component name="ledControl">
    <implementation.remora
              implementer="cmu.telosb.peripheral.Leds"/>
  </component>
  <component name="blink">
    <implementation.remora implementer="app.BlinkApp">
  </component>
  <component name="timer"/>
    <implementation.remora implementer="core.sys.Timer"/>
  </component>
  <!—components wiring -->
  <wire source="blink/iLeds" target="ledControl/iLeds"/>
</composite>
```

Figure 9.9: Blink application configuration.

In the next phase, the REMORA engine reads the elements of the development box and also OS libraries in order to generate the REMORA framework including the source code of components and OS-support code. Then, application object file will be created through OS-provided facilities and finally deployed on sensor nodes.

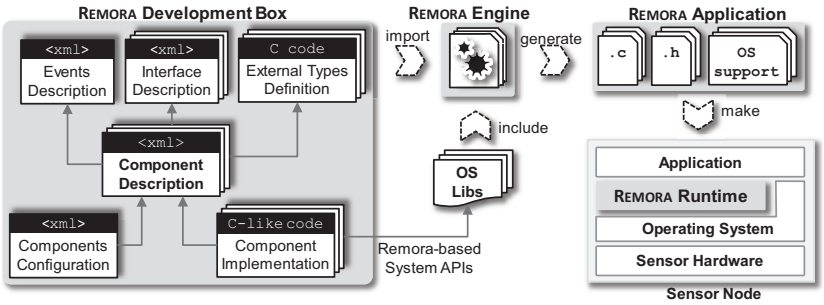

Figure 9.10: REMORA-based development process.

9.2.5 Middleware Programming

The research efforts on sensor middleware have hitherto focused on developing services and algorithms for routing, quality of service, energy-efficiency, resource management, localization, synchronization, etc. These, however, often fall short of expectation in integrating services and algorithms into a generic middleware system, and in helping application programmers to compose a system that exactly matches their requirements. This raises the need for a specific approach for middleware programming in WSNs that goes beyond dealing with only application-specific logics. From the programming point of view, middleware

services are distinguished from other components in the system by the following two main factors.

First, despite the application-level programming, middleware components are developed very close to the operating system, requiring to tightly interact with system-level components. Therefore, sensor programming models, supporting middleware development, should provide the primitives required to interface between middleware services and system components. REMORA addresses this concern through the OS Abstraction Layer and the OS-Wrapper components. In addition to enabling the portability of sensor applications, these principles make REMORA a suitable programming model to build middleware applications.

Second, middleware solutions should be exposed as a well-packaged, stand-alone application which can be easy integrated to the target application with minimum programming effort. Although this issue has been extensively addressed in conventional resource-rich systems, software pieces in WSNs are often assembled together in an ad-hoc manner, without any well-established software composition model. This problem originates from the fact that WSN programming abstractions do not pay enough attention to software composition and integration approaches. With the increasing number of intermediate software solutions for WSNs (*e.g.*, networking, algorithms and QoS), programming constructs are required to compose the application, middleware services, and the operating system into a unified sensor software in a generic, simple and robust manner.

The technique we have adopted in REMORA to compile and assemble components has the potentials to meet a higher level of assembly which is *integrating a given set of* REMORA*-based applications*. In particular, we enhance the REMORA engine with the capability of processing multiple isolated REMORA applications and integrating them into a unified system. The main concerns, in this endeavor, include how to expose an application's functionality as an API and bind applications based on the dependencies between their APIs. REMORA addresses these concerns based on the concept of *Autonomous Composable Module* (ACM). This refers to developing REMORA applications in an autonomous manner so that the programmer considers an under-development application as a stand-alone module with its own operations. It means that, based on this approach, the dependencies of the application to others are not declared within its description. The REMORA engine is in charge of analyzing dependencies among ACMs and binding them together. Figure 9.11 shows the overall architecture of REMORA composition solution, consisting of a set of ACMs and the main sensor application. The latter not only implements the application logic, but also serves as a starting point to execute programs. An ACM contains a set of REMORA components implementing its logics, as well as a component representing its API.

As a use case for the REMORA middleware programming model, in [15] we demonstrate a run-time middleware system, called REMOWARE, to support dynamic reconfiguration of REMORA-based applications. REMOWARE is basically an ACM which can be easily used as a middleware solution in any dynamic sensor application to enable run-time reconfiguration of REMORA components.

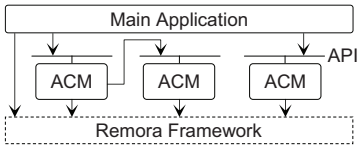

Figure 9.11: The overall architecture for composing the main application and ACMs.

9.2.6 Automatic Tuning

Besides the componentization of application-level modules, REMORA can be exploited to componentize operating system's modules either by wrapping them in REMORA components, or redeveloping them according to the REMORA specification. This enables the REMORA engine to expand its control on the configuration of sensor software and therefore makes it possible to automatically tune the target software installed on nodes. In this way, the REMORA engine can gain a meta knowledge showing which OS-level components are involved in supporting application logic and based on that it can trace the interactions between application components and system components. In this way, it can identify the *orphan* components—the components that are not involved in the application scenario execution.

Figure 9.12 describes an *initial* configuration (prior to deploying on nodes) of sensor software in which the application-level components gain system services through OS-wrapper components at the runtime layer. These components interact either directly with kernel-level modules, or with other intermediary wrapper components beneath the runtime. This initial setting can be optimized by REMORA engine. When it executes the tuning process, deduces that one of the intermediary components is orphan, and removes it from the final package installed on nodes.

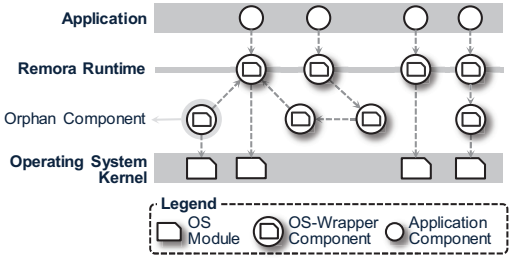

Figure 9.12: The REMORA engine tunes the operating system by tracing component dependencies and finding orphan components.

9.3 Implementation

To discuss the implementation of REMORA, we structure this section according to the main modules proposed for REMORA-based application development, namely, the *engine*, the *framework*, and the *runtime*. Since the platform supporting the component model is comprehensive and includes numerous implementation issues, we only highlight the key technologies and design techniques used for implementing each of the aforementioned modules. Beyond the internal design of modules, the overall design goal is to keep the artifacts of each module completely independent from others in the sense that in the final system, each module is composed of three set of source codes dedicated to corresponding modules. The main advantage of this separation is to minimize the required effort to port the component model to a new operating system by ensuring a clear isolation between the REMORA framework and the REMORA runtime.

9.3.1 Remora Engine

The REMORA engine is deployed on the programmer's desktop machine to read all artifacts within the development box, perform required analyses for code generation, and generate the final C code of components, as well as OS-support code. We adopt Java to develop the engine because of its cross-platform capabilities, as well as its strong support for XML processing. Additionally, the object-oriented nature of Java simplifies the complex process of code analyzing and code generation. We briefly discuss the key design principles of this Java-based engine below.

The first task of the engine is to parse the C-like implementation of components and extract the information concerning the specification of REMORA. To this end, we have developed a parser module, which is originally generated by ANTLR—a widely used open-source parser generator [16]. Since this generated tool only parses the source code, we have modified the generated parser to extract REMORA-required information, such as *name*, *signature*, and *body* of implementation functions. By doing that, the engine builds a meta-data structure containing all required information about the implementation of a component and the rest of the engine tasks are performed based on that.

The other key implementation part of the REMORA engine deals with *processing events*, *component instantiation*, and *component lifecycle*. This unit deduces the multiplicity type of components according to the algorithm 1 and generates the necessary data structures. This algorithm determines the multiplicity type based on the type of events generated by the component, as well as whether the component owns any property or not. If the final value of variable $InstNumber$ is 0, this means that the component has no instance and only requires the code memory, while the value of 1 shows that only one instance of component's data should be stored in the data memory. Finally, for a multiple instance component the value of $InstNumber$ is 2.

This module also features a set of well-defined techniques, such as *in-component call graph analyzer* and *cross-component call tracker* to support stateful component. The for-

Algorithm 1 Determining the multiplicity type of components

Input: *producedEvents*, events generated by the components
Input: *properties*, component's properties
Output: component's multiplicity type

```
InstNumber ⇐ −1
MultiConsumers ⇐ false
for aEvent in producedEvents do
  if aEvent is unicast then
    if sizeOf(aEvent.consumers) > 1 then
      MultiConsumers ⇐ true
      break
    end if
  end if
end for
if MultiConsumers is false then
  if sizeOf(producedEvents) > 0 then
    InstNumber ⇐ 1
  else
    if sizeOf(properties) > 0 then
      InstNumber ⇐ 1
    else
      InstNumber ⇐ 0
    end if
  end if
else
  InstNumber ⇐ 2
end if
```

mer concept is concerned with discovering *state-dependent* functions of a component. Two types of state dependency can be envisaged for a function: *i) explicit dependency*: the component's property(s) is(are) directly accessed within the function's code, *ii) implicit dependency*: the function contains direct/indirect invocation(s) to an explicit type. To preserve the state of a component, we need to retain a pointer to the component's context and pass it to the state-dependent functions of component. The in-component call graph analyzer employs a recursive technique to navigate the function calls with the component and identify the state-dependent functions. Likewise, the cross-component call tracker tracks the interactions between components in order to retain the state of components. Finally, the major task of the engine is to support events and manage the component lifecycle by embedding framework-support patches in the component implementation.

Automatic tuning of sensor software is the other responsibility of the REMORA engine. The data structure supporting the tuning process is a directional graph in which every node represents a component of the system and edges between nodes are the service-based interactions among the components (cf. Figure 9.13). The engine first creates this graph and then navigates the nodes based on the Depth-First Search (DFS) algorithm to find the orphan nodes. In particular, it initiates this process from the main component of application, implementing the interface `ISensorApp`, as the root of graph. When it accomplishes DFS, it removes orphan nodes—all components that are never visited by DFS.

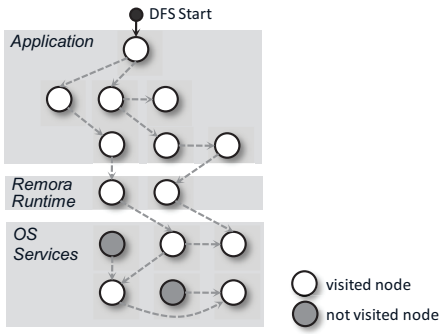

Figure 9.13: Using depth-first search algorithm to discover the orphan nodes.

Moreover, the REMORA engine undertakes binding ACM modules in order to support middleware programming. This process is carried out in a two-phase strategy. It first processes the components configuration document of each ACM and creates a disconnected, directed graph structure in which each ACM would have directed edges to the required APIs. In the second phase, the engine analyzes the yielded disconnected graph from the first phase and creates a connected graph representing dependencies among ACMs, as well

as between the main sensor application and ACMs. Therefore, it provides a higher-level of wiring model between co-habiting applications and this model is further processed by the engine to implement the execution flow graph in the system.

9.3.2 REMORA Framework

The REMORA framework is composed of a collection of core C programs, supporting the event management model of REMORA and hosting the target application's components. As mentioned before, the REMORA framework is an OS-independent module. There are two main reasons for this: *i)* the core of the framework is written in the C language and also the final code of application's components are translated to equivalent C programs by the REMORA engine, *ii)* the framework is linked to the OS via the REMORA runtime which translates all OS-originated interactions (*e.g.*, OS-events) to a set of pre-defined, application-specific instructions understandable by the framework (cf. Section 9.2.3). The other possible dependency issue is caused by the mechanism used to form the REMORA framework as a *process* within the OS and *schedule* it to run. This is also extensively addressed by the REMORA runtime as explained in Section 9.3.3.

The main mission of the framework is to facilitate event management tasks, including *scheduling* and *dispatching*. To explain these tasks, we first introduce two *queue* data structures supporting our event model. The first queue is dedicated to the *event producer* components (PQ), while the second one is designed to maintain the *event consumers* (CQ). We discuss here how the REMORA framework is built based on these data structures.

Scheduling in REMORA refers to all operations required to *enqueue* and *dequeue* event producers and event consumers. In particular, the main concern is *when* to enqueue/dequeue a component and *who* should perform these tasks. The REMORA framework addresses these issues based on the observation model of events. For example, if an event is *automatically* observable, the associated producer component and all the subscribed consumers are enqueued by the framework core during the application startup, while in a *manual* observation, producer and consumer are placed respectively in PQ and CQ when the consumer component calls the `start` function of observation interface. A question may arise is that prior to initiating the scheduling mechanism, how the components instances are created. In REMORA, memory allocation for components is done statically. Therefore, the memory address of all instances of all components are determined during the framework compilation and we do not impose the high overhead of dynamic memory allocation to such a resource-constraint platform. At runtime, parts of the framework, embedded in each component, are responsible for dealing with component lifecycle—*e.g.*, activating or deactivating event generator components.

The other role of the REMORA framework is to periodically poll the generator components for event observation, and then feed event handlers with the matched events. To achieve the former, event generators in REMORA keep a pointer to the globally known callback function, *dispatcher*, thereby, the REMORA framework is able to poll event gener-

ators by periodically calling this function. Similarly, the latter is realized by invoking the callback handler function within the event consumer component like `timerExpired` in the Blink component.

Figure 9.14 illustrates the *dispatching* mechanism in the framework including the supporting data structures. In *Polling*, the REMORA framework continuously polls the EventProducer components through *dispatcher*—the globally known callback function. Whenever a producer dispatches an event (AbstEvent), the framework casts this event to the actual event type, which is either UCastEvent(unicast event) or MCastEvent(multicast event). UCastEvent will be directly forwarded to the subscribed consumer through the callback function pointer stored in the UCastEvent. If a MCastEvent is generated, the framework delivers it to all the interesting components formerly enqueued. For OS-events, the same procedure is followed except the polling phase, which is performed by the operating system.

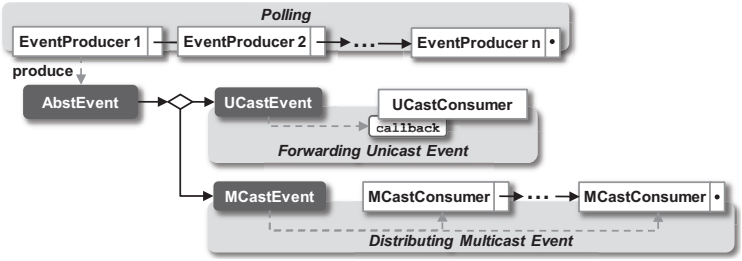

Figure 9.14: REMORA event processing mechanism.

9.3.3 REMORA Runtime

The REMORA framework is integrated with the underlying operating system through the REMORA runtime. In our current implementation, the core of the REMORA runtime is a Contiki-compliant *process* running together with all other *autostart* processes of Contiki (see Figure 9.15). This process undertakes two tasks: *i)* periodically scheduling the REMORA framework (for polling event generator components) to run, and *ii)* listening to the OS-events and delivering the relevant ones to the REMORA framework. By relevant, we mean the REMORA runtime recognizes those OS-events that are of interest to the application. To achieve such a filtering, the source code of this part is generated by the REMORA engine according to the events description of target application and then imported to the REMORA runtime. By doing that, we provide a lightweight event dissemination mechanism interpreting only application-specific OS-events.

In addition, the application code may need to use libraries available in the OS. In REMORA, a programmer can develop a set of REMORA components acting as system API providers. In fact, these components delegate all high-level system calls to the correspond-

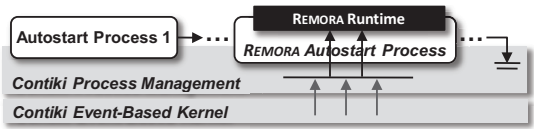

Figure 9.15: Integration of Contiki and REMORA through the runtime layer.

ing OS-level functions—*e.g.*, the `currentTime()` function call in the system API is delegated to the Contiki function `clock_time()`. We offer this API to decouple the application components from OS modules and ensure the portability of REMORA-based applications. If an application is not expected to be ported to other operating systems, programmers can directly call the OS functions within component code and therefore slightly improve the runtime performance.

9.4 Evaluation

To evaluate the efficiency of REMORA, in this section we first demonstrate and assess a real REMORA-based application, then we focus on the general performance figures of REMORA.

9.4.1 A Real Remora-based Deployment

Our real application scenario is a network-level *application suite* consisting of a set of mini applications bundled together. This suite is basically designed to provide services, such as *code propagator* and *web facilities* in WSNs. We focus here on the first one and design it based on the REMORA approach.

Code propagation becomes a very important need in WSNs when we need to update remotely the running application software [17]. The code propagator application is responsible for receiving all segments of a running application's object code over the network and loading the new application image afterwards. The code propagator exploits the TCP and UDP protocols to propagate code over the network. At first, TCP is used to transfer new code, block by block, to the sink node connected to the code repository machine, and then UDP is used to broadcast wirelessly new code from a sink node to other sensor nodes in the network. When all blocks are received, the code propagator loads the new application.

Figure 9.16 describes the components involved in the first part of our application scenario. TCPListener is a core component listening to TCP events. This multiple-instances event generator is created for each TCP event consumer component with unique listening port number. For example, CodePropagator receives data from port 6510 (`codePropPort`), while WebListener is notified for all TCPEvents on port 80 (`webPort`). CodePropagator stores all blocks of new code in the external flash memory through the interface `IFile` implemented by the FileSystem component. When all blocks are received, CodePropagator loads the new

application by calling the interface `ILoader` from the component ELFLoader. These two interfaces are system APIs that delegate all application-level requests to the OS-specific libraries. The interface `INet`, implemented by the component Network, is also the other system API providing the low-level network primitives to TCPListener.

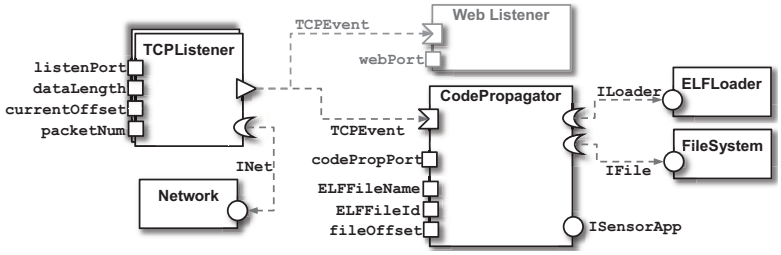

Figure 9.16: Code propagation application architecture.

As mentioned before, we adopt Contiki as our OS platform to assess the Remora component model. Contiki is being increasingly used in both academia and industrial applications in a wide range of sensor node types. Additionally, Contiki is written in the standard C language and hence Remora can be easily ported to this platform. Finally, the great support of Contiki on event processing and process management motivate us to design and implement the Remora runtime on this OS. Our hardware platform is the popular TelosB mote equipped with a 16-bit TI MSP430 MCU with $48KB$ ROM and $10KB$ RAM.

The concrete separation of concerns in this application is the first visible advantage of using Remora. The second improvement is the *easy* reuse of TCPListener for other TCP-required applications, which is not the case in a non-componentized implementation. In particular, for each new application, we only need to instantiate the *context* of TCPListener and configure its properties (like port number) accordingly—*e.g.*, WebListener in Figure 9.16.

Memory Footprint. Table 9.1 reports the memory requirement of Remora and Contiki programming model (*protothreads*) for implementing the code propagation application. As indicated in the table, the Remora-based development does not impose additional data memory overhead, while it consumes extra 532 bytes of code memory, which is essentially related to the cost of framework and runtime modules. This cost is paid once and for all, regardless of the size and the number of applications running on the sensor node. The code memory cost could be even further reduced by removing system APIs (Network, FileSystem, and ELFLoader) and calling directly the Contiki's libraries within CodePropagator. Note that the overhead of TCPListener can also be decreased when this component is shared for the use of other applications—*e.g.*, WebListener. Therefore, we can conclude that the memory overhead of Remora is negligible compared to the high-level features it provides to the end-user.

Table 9.1: The memory requirement of code propagation application in REMORA-based and Contiki-based implementations.

Programming Model		Code Memory (bytes)	Data Memory (bytes)
Contiki		**722**	**72**
REMORA	Code Propagation Components		
	CodePropagator	252	36
	TCPListener	310	0
	System API Components		
	ELFLoader	38	0
	Network	92	0
	FileSystem	68	0
	REMORA Core		
	Framework and Runtime	494	14
	Total	**1254**	**50**
Remora overhead		**+532**	**-22**

Processing Cost. Figure 9.17 reports the comparison of CPU costs in these two approaches. The time measurement was started when the first block of new application's code was received and it was stopped when the last block of code arrived to the sensor node. Since in-file seeking and writing is a costly process, we removed invocations related to FileSystem and ELFLoader and measured the execution time afterwards. As the size of new code (ELF file) is increased, the processing overhead of REMORA is also slightly increased compared with the equivalent Contiki implementation. We believe that this very low overhead is due to the extra context-switchings (among event processing functions within the REMORA runtime) occurring for larger code in REMORA, which is not the case in the Contiki-based implementation.

Programming Effort. Evaluating the programming effort is difficult since it is affected by factors difficult to measure—*e.g.*, the nature of code (algorithmic or routine), the complexity of the processing, and syntax and semantic of programming languages. However, WSN programming research has hitherto adopted the number of *lines of code* (LOC) as a simple indication. Table 9.2 reports this metric for the two main components of code propagator application. It is interesting to compare these measurements against the equivalent functionality available in Contiki libraries, where it is directly developed atop of the operating system. The Contiki-based implementation of the TCP listener module contains 41% more LOC than our version. This efficiency is achieved since in our implementation event-handling code is embedded in the run-time system and shared for the use of different applications. We also gain a significant improvement in LOC for code propagator module compared with the Contiki's implementation. It is because the verbose code of event handling in Contiki programming model is replaced with the shortened C-like code

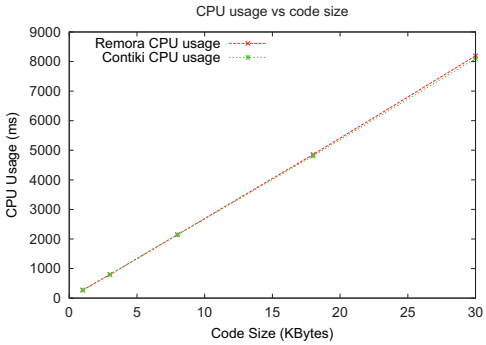

Figure 9.17: CPU usage for receiving new code by propagator application in REMORA and Contiki.

Table 9.2: Line of code for our main components.

Component	Line of Code (Remora)	Line of Code (Contiki)	Reduced Effort
TCP Listener	62	104	**41%**
Code Propagator	19	36	**47%**

of REMORA.

Tuning Result. The efficiency of the tuning technique directly depends to the target use case and its requirements in terms of low-level system services. In the case of the code propagation application, we cannot precisely measure the reduction of the final object code size as it is basically an intermediate application lying beneath the main sensor application. Therefore, we measure the tuning performance of the code propagator by considering it as a main sensor application. Applying tuning technique on this application yields 5% reduction in the final Contiki binary object file. This efficiency is achieved by automatic removal of modules that never involve in the code propagation process, *e.g.*, programs interfacing a node's peripherals (*e.g.*, light, button and sensors).

The rest of this section is devoted to the assessment of two main performance figures of REMORA, namely, memory footprints and CPU usage.

9.4.2 Memory Footprint

In REMORA, we have made a great effort to maintain memory costs as low as possible. The first step of this effort is to avoid creating meta-data structures, which are not beneficial

in a static deployment. Distinguishing unicast events and multicast events has also led to a significant reduction in memory footprints as REMORA does not need to create any supporting data structure for unicast events.

The memory footprints in REMORA is categorized into a minimum overhead and a dynamic overhead. The former is paid once and for all, regardless of the amount of memory is needed for the application components, while the latter depends on the size of application. Table 9.3 shows the minimum memory requirements of REMORA, which turn out to be quite reasonable with respect to both code and data memory. As mentioned before, our sensor node, TelosB, is equipped with $48KB$ of program memory and $10KB$ of data memory. As Contiki consumes roughly $24KB$ (without µIP support) of both these memories, REMORA has a very low memory overhead considering the provided facilities and the remaining space in the memory.

Table 9.3: The minimum memory requirement of REMORA.

Module	**Code Memory (bytes)**	**Data Memory (bytes)**
Framework Core	374	4
Runtime Core	120	10
Total	**494**	**14**

Table 9.4 shows the memory requirement of different types of modules in the REMORA framework. The exact memory overhead of REMORA depends on how an application is configured, *e.g.*, an application, containing one single instance event producer and one unicast event, needs extra 56 bytes $(38 + 8 + 10)$ of both data and code memory. Ordinary components do not impose any memory overhead as REMORA does not create any meta data structures for them. For other types of modules, REMORA keeps the data memory overheads very low as this memory in our platform is really scarce. We also believe that the code memory overhead is not significant since a typical WSN application is small in size and it may contain up to a few tens of components, including ordinary components. It should be noted that componentization itself reduces the memory usage by maximizing the reusability degree of system functionalities like the one discussed in the code propagation application.

9.4.3 CPU Usage

As energy cost of REMORA core is limited to only the use of the processing unit, we focus on the processing cost of our approach and show that REMORA keeps the CPU usage at a reasonable level, and in some configurations it even reduces CPU usage compared to the Contiki-based application development.

To perform the evaluation, we set up a Blink application in which a varying number of

Table 9.4: The memory requirement of different entities in REMORA.

Entity		**Code Memory (bytes)**	**Data Memory (bytes)**
Ordinary Component		0	0
Event Producer	Single Ins.	38	8
	Multiple Ins.	42	10
Event	Unicast	0	10
	Multicast	0	10
Multicast Event Consumer		30	6
OS Event		28	4
System API		4	0

mirror components (1 to 15) switch LEDs on and off every second. The two implementations of this application, Contiki-based and REMORA-based, were compared according to a CPU measurement metric. The metric was to measure the amount of time required by one REMORA component and one Contiki process to switch LEDs six times: three times on and three times off. With the less number of switches, we cannot extract the exact timing differences as our hardware platform provides a timing accuracy of the order of one millisecond.

We started our evaluation by deploying an application similar to the one presented in Section 9.2.1 and measuring the CPU usage based on our metric. In each next evaluation step, we added a mirror Blink component to the application and measured again the time. This experiment was continued for 15 times. We made the same measurement for a Contiki-based Blink application and added a new Contiki Blink process in each step. Figure 9.18 shows the evaluation result of our scenario. When we have one Blink component/process, the CPU overhead of both approaches is almost the same, indicating that the REMORA runtime and framework impose no additional processing overhead. When the number of components/process increases towards 15, reduction in CPU usage is achieved in two dimensions.

Firstly, the number of CPU cycles for REMORA is slightly less than for the Contiki application. This difference reaches 13 milliseconds when Contiki undertakes running 15 Blink processes. Therefore, we can conclude that REMORA does not impose additional processing overhead affecting the performance of the system. Secondly, the CPU usage of REMORA application is reduced when the number of Blink components is increased. This improvement is achieved because the number of context switches between the REMORA runtime and the REMORA framework is significantly decreased when there are more event producer components (Timer) in PQ.

To clarify this issue, we assume that the application running time is T and Contiki periodically allocates CPU to the REMORA runtime in this period. In each allocation round, the runtime module invokes the event manager in the REMORA framework to poll

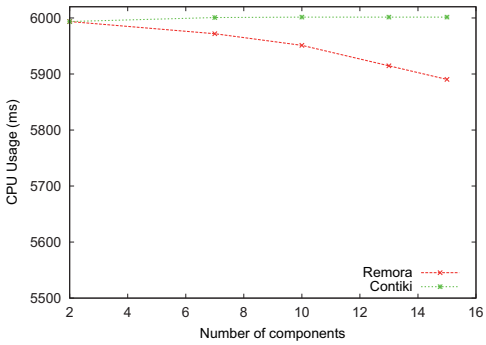

Figure 9.18: The REMORA-based implementation does not impose additional CPU overhead compared to the Contiki-based implementation.

the application level event producers. Given that there are K producers in PQ, the polling process consumes $K \times t_1$ of CPU, where t_1 is the average processing cost of one element. Therefore, the frequency of event manager calling (equal to the number of context-switches) is in the order of $T/K \times t_1$. Therefore, as the value of K is increased the number of context-switches is decreased accordingly. Figure 9.19 shows the changes in the number of context-switches when the number of Timer components is increased to 15. As a result, the maximum performance in REMORA relies on the average number of event producer components enqueued during the application lifespan, while in the worst case (a very few producers in the queue) REMORA does not impose any additional processing cost.

9.5 Existing Approaches

In this section, we survey the existing component-based approaches for programming on embedded system and WSNs. As mentioned before, a number of these component models are proposed not only to facilitate development of application modules, but also to build component-based operating systems for WSNs. Furthermore, the other objective behind component-based frameworks for WSNs has been the provision of run-time reconfigurability in dynamic WSN applications. There are also a few attempts devoted to porting the existing component-based approaches to other platforms—*e.g.*, embedded systems, large-scale systems, to sensor platforms with some minor changes.

NESC [8] is perhaps the best known component model being designed specifically for WSNs and used to develop TINYOS [18]. Knowing NESC language, programming in TINYOS is quite simple and the developed components are reusable in different applications. As mentioned earlier, the main downside of NESC is that it is tightly bound to the

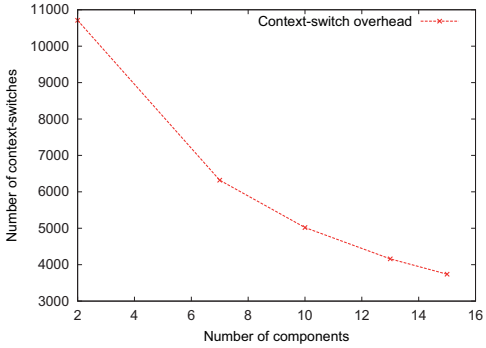

Figure 9.19: As the number of producer components in the queue is increased, the number of context switches is significantly decreased.

TinyOS platform. Moreover, although NesC efficiently supports event-driven programming, events in NesC are not considered as independent entities with their own attributes and specifications. Therefore, the binding model of event-related components is not well-described as it is not essentially described based on the specification of events. Additionally, the unique features of Remora, such as multiplicity in component instance and property-based reconfiguration of components bring significant improvements to component-based programming in WSNs compared to NesC.

Coulson et al. in [9] propose OpenCom as a generic component-based programming model for building system applications without dependency on any target-specific platform environment. The authors express that they have tried to build OpenCom with negligible overhead for supporting features specific to a development area, however it is a generic model and basically developed for platforms without resource constraints and tends to be complex for embedded systems.

To evaluate OpenCom, we deployed a sample *beacon* application [19], including Radio, Timer and Beacon components, on a TelosB node with Contiki. Based on our measurements, the memory footprint of this application is significantly high, so that it consumes 4,618 bytes of code memory and 28 bytes of data memory. As a real application, Gridkit [20] is an OpenCom-based middleware for sensor networks, realizing co-ordinated distributed reconfigurations based on policies and context information provided by a context engine. This middleware was deployed on Gumstix-based [21] sensor platforms (a resource-rich node type) for a flood-monitoring scenario, where the minimum memory requirement of Gridkit core middleware and OpenCOM run-time is about 104 KB of memory. Lorien [22] is an OpenCom-driven approach that was recently proposed to provide a fully reconfigurable OS platform in WSNs, however this work is still at an initial stage of devel-

Table 9.5: Overview of existing component-based approaches to WSN programming.

Approach	OS Platform	Core Size(KB)	Cost per Component (Bytes)
LORIEN	LORIEN	5.5	350
THINK	OS-Indep.	2	102
FIGARO	CONTIKI	2	15
LOOCI	SUNSPOT	20	587
Remora	**OS-Indep.**	**0.5**	**8**

opment.

FIGARO [23] is a WSN-specific dynamic component model, focusing on *what* and *where* should be reconfigured. Specifically, Figaro proposes a set of C macros representing a new component model exploitable over any operating system written in the C language. However, the dynamic aspect of FIGARO—its main feature—is only exploitable on the Contiki operating system. Apart from that, FIGARO fails to consider event management issues at the component design level and mostly relies on the operating system's event handling features.

LOOCI [24] is a component-based approach, providing a loosely-coupled component infrastructure focusing on an event-based binding model for WSNs, while the Java-based implementation of LOOCI limits its usage to the SUNSPOT sensor node.

The THINK framework [10] is an implementation of the FRACTAL [25] component model applied to operating systems. The choice of the THINK framework is motivated by the fact that it allows fine-grained reconfiguration of components. Although the experiments on deploying THINK components on WSNs have been quite promising in terms of memory usage [26], the lack of application-level event support is the main hurdle for using THINK in WSNs.

Table 9.5 shows a summarized comparison of REMORA with other works proposed in this category in terms of minimum memory required for the core and additional memory overhead per component.

The OSGi model [27] is a framework targeting powerful embedded devices, such as mobile phones and network gateways along with enterprise computers. OSGi features a secure execution environment, support for runtime reconfiguration, lifecycle management, and various system services. While OSGi is suitable for powerful embedded devices, the smallest implementation, Concierge [28] consumes more than $80KB$ of memory, making it inappropriate for resource-constrained platforms.

OSKIT [29] is a set of off-the-shelf components for building operating systems. OSKIT is developed with a programming anguage called KNIT [30]. However, in contrast to NESC, KNIT is not limited to OSKIT. Nevertheless, OSKIT has adapted the Microsoft COM model and is not primarily focused on embedded systems.

9.6 Discussion: Extension Opportunities

We believe that the current specification of REMORA along with its low resource requirements can tackle the concerns we mentioned at the beginning of this paper. However, there are a number issues—to further support advanced programming in WSNs—that has not been considered by the current REMORA yet. In this section, we focus on these issues and identify potential solutions.

Dynamic Reprogramming. Enabling dynamic reprogramming in WSNs becomes a vital feature when the target application is subject to changes—*e.g.*, fixing bugs, upgrading operating system and applications, and adapting applications behavior according to the physical environment [31, 32, 17]. Although the component-based nature of REMORA can simplify the support for dynamic replacement of system modules, the restrictions on the REMORA component model, including the lack of dynamic memory allocation and the absence of a meta-data to dynamically handle the interactions between components, make the reconfiguration of REMORA components a challenging issue. In fact, the main problem is that how to efficiently provide such a feature in such a way that the overhead of dynamic memory allocation is carefully minimized. Reducing the additional memory required to store the meta-data is another issue in the way of upgrading REMORA to a dynamically reconfigurable module.

Componentization of an OS using Remora. As mentioned earlier, the current goal of REMORA is to be exploited only in application-level programming. However, we believe that the efficient support of event processing in REMORA potentially enables it to componentize system level functionalities. This can also increase the customization of an operating system for a particular WSN application. In the Blink application, we implicitly demonstrated this capability by wrapping the Timer component, which is essentially developed at the OS level. To address precisely this issue, we need to enhance the current REMORA implementation with features like *concurrency support*, *task scheduling*, and *interrupts handling*.

Supporting Preemption. In our current implementation, a REMORA process cannot be preempted by any other process in the operating system. This issue becomes critical when a component execution takes a long time to complete and it causes large average waiting times for other processes waiting for the processor. The event handling model of REMORA can be used to provide preemption by defining a new event type per preemption-required point of application, while in this case the component implementation and the event management become quite complicated. This concern will also be considered in the future extensions for REMORA. In particular, we intend to promote the native Contiki macros, handling process lifecycle, to the REMORA application level. In this way, the REMORA component becomes preemptable by explicitly yielding the running process.

Distribution Support. Beside the fact that REMORA provides a strong abstraction for single node programming, the same level of programming abstraction is expected to occur at the network level. This challenge opens up another key area for future work: how to make REMORA components distributed by the provision of a well-defined remote invocation

mechanism. In particular, this refers to rather programming with low-level APIs to provide distribution; we can automatically generate the code which is required for sending data over the network or invoking methods. As a result, the communication strategy could be reified at the architecture level and therefore relieve the programmer from dealing with the specificities of the protocol she/he will need to use for exposing her/his services across the network.

9.7 Conclusions

From a high-level programming point of view, WSNs are still difficult to program. Most of the state-of-the-art programming approaches address this issue by slightly extending low-level system programming languages and promoting them as a solution for application development in WSNs. In this article, we considered WSN high-level programming as a challenge independent from low-level programming paradigms and presented Remora as a novel programming abstraction for resource-constrained embedded systems.

Remora simplifies high-level event-driven programming in WSNs by a component-based approach portable to different operating system platforms. Involving PC-based developers in WSN programming and conforming Remora to the state-of-the-art technologies for component development are two other challenges addressed in this article. The special consideration paid to the event abstraction in Remora makes it a practical and efficient approach for WSN applications development. The other key features of Remora include: simplifying middleware services development, enabling tunability of operating system software by wrapper components, rich support of component reusability and instantiation, and reduced effort and resource usage in WSN programming.

Careful restrictions on the Remora component model, including the lack of dynamic memory allocation and avoiding M-to-N communications between event producers and event consumers bring significant improvements to the static deployments in WSNs, where the main improvement happens in sensor memory usage. The main additional memory overhead is induced by the Remora runtime, occupying only 1% of the total code memory on our sensor platform, which is a very low overhead considering the provided facilities and the remaining space in the memory.

The remora future work targets all issues discussed in the previous section. In particular, we are currently considering the first issue and investigating how the Remora specification should be modified to support dynamic programming in WSNs with a reasonable cost.

References

[1] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, pp. 19:1–19:51, 2011.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Boston, MA, USA: Addison-Wesley, 2002.

[3] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Technical concepts of component-based software engineering," Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, USA, Tech. Rep. CMU/SEI-2000-TR-008, May 2000.

[4] R. Van Ommering, F. Van der Linden, J. Kramer, and J. Magee, "The koala component model for consumer electronics software," *Computer*, vol. 33, no. 3, pp. 78–85, 2000.

[5] T. Genssler, A. Christoph, M. Winter, O. Nierstrasz, S. Ducasse, R. Wuyts, G. Arévalo, B. Schönhage, P. O. Müller, and C. Stich, "Components for embedded software: the pecos approach," in *Proc. of the Int. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*. Grenoble, France: ACM, 2002, pp. 19–26.

[6] H. Hansson, M. Akerholm, I. Crnkovic, and M. Torngren, "Saveccm - a component model for safety-critical real-time systems," in *Proc. of the 30th EUROMICRO Conf.* Washington, DC, USA: IEEE Computer Society, 2004, pp. 627–635.

[7] A. Plšek, F. Loiret, P. Merle, and L. Seinturier, "A component framework for java-based real-time embedded systems," in *Middleware '08: Proc. of the 9th ACM/IFIP/USENIX Int. Conf. on Middleware*. Leuven, Belgium: Springer-Verlag, 2008, pp. 124–143.

[8] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesc language: A holistic approach to networked embedded systems," in *PLDI '03: Proc. of the ACM SIGPLAN 2003 Conf. on Programming language design and implementation*. San Diego, California, USA: ACM, 2003, pp. 1–11.

[9] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1–42, 2008.

[10] J.-P. Fassino, J.-B. Stefani, J. L. Lawall, and G. Muller, "Think: A software framework for component-based operating system kernels," in *ATEC '02: Proc. of the General Track of the USENIX Annual Technical Conf.* Berkeley, CA, USA: USENIX Association, 2002, pp. 73–86.

[11] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. L. Trung, and F. Eliassen, "Programming sensor networks using REMORA component model," in *DCOSS '10: Proc. of the 6th Int. Conf. on Distributed Computing in Sensor Systems*. Santa Barbara, CA, USA: Springer, 2010, pp. 45–62.

[12] OSOA, "The service component architecture," http://www.oasis-opencsa.org/sca.

[13] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proc. of the 29th Annual IEEE Int. Conf. on Local Computer Networks*. Tampa, Florida, USA: IEEE Computer Society, 2004, pp. 455–462.

[14] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, "Protothreads: simplifying event-driven programming of memory-constrained embedded systems," in *SenSys '06: Proc. of the 4th Int. Conf. on Embedded networked sensor systems*. Boulder, Colorado, USA: ACM, 2006, pp. 29–42.

[15] University of Oslo, "The Remora Component Model," 2010, http://folk.uio.no/amirhost/remora.

[16] ANTLR, "Parser Generator," http://www.antlr.org.

[17] B. Pásztor, L. Mottola, C. Mascolo, G. P. Picco, S. A. Ellwood, and D. W. Macdonald, "Selective reprogramming of mobile sensor networks through social community detection," in *Proc. of 7th European Conf. on WSNs (EWSN)*, vol. 5970. Coimbra, Portugal: Springer-Verlag, 2010, pp. 178–193.

[18] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for sensor networks," in *Ambient Intelligence*. Berlin, Germany: Springer Verlag, 2004, pp. 15–148.

[19] I. Chatzigiannakis, S. Fischer, C. Koninis, G. Mylonas, and D. Pfisterer, "WISEBED: an open large-scale wireless sensor network testbed," in *SENSAPPEAL '09: Proc. of the 1st Int. Conf. on Sensor Networks Applications, Experimentation and Logistics*, ser. Lecture Notes of the Institute for Computer Sciences. Athens, Greece: Springer-Verlag, 2009, pp. 68–87.

[20] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes, "Dynamic reconfiguration in sensor middleware," in *MidSens '06: Proc. of the Int. Workshop on Middleware for sensor networks*. Melbourne, Australia: ACM, 2006, pp. 1–6.

[21] GUMSTIX, "Gumstix embedded computing platform specifications," 2004, http://www.gumstix.com.

[22] B. Porter and G. Coulson, "Lorien: a pure dynamic component-based operating system for wireless sensor networks," in *MidSens '09: Proc. of the 4th Int. Workshop on Middleware Tools, Services and Run-Time Support for WSNs*. Illinois: ACM, 2009, pp. 7–12.

[23] L. Mottola, G. P. Picco, and A. A. Sheikh, "Figaro: fine-grained software reconfiguration for wireless sensor networks," in *EWSN '08: Proc. of the 5th European Conf. on WSNs*. Bologna, Italy: Springer-Verlag, 2008, pp. 286–304.

[24] D. Hughes, K. Thoelen, W. Horré, N. Matthys, P. J. del Cid Garcia, S. Michiels, C. Huygens, and W. Joosen, "Looci: A loosely-coupled component infrastructure for networked embedded systems," in *Proc. of the 7th Int. Conf. on Advances in Mobile Computing & Multimedia*. Kuala Lumpur, Malaysia: ACM, Dec. 2009, pp. 195–203.

[25] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java: Experiences with auto-adaptive and reconfigurable systems," *Softw. Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[26] O. Lobry, J. Navas, and J.-P. Babau, "Optimizing component-based embedded software," in *COMPSAC '09: Proc. of the 33rd Annual IEEE Int. Computer Software and Applications Conf.* Washington, DC, USA: IEEE Computer Society, 2009, pp. 491–496.

[27] OSGi Alliance, "The OSGi framework," 1999, http://www.osgi.org.

[28] J. S. Rellermeyer and G. Alonso, "Concierge: a service platform for resource-constrained devices," *ACM SIGOPS Operating Systems Review*, vol. 41, no. 3, pp. 245–258, 2007.

[29] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers, "The flux oskit: a substrate for kernel and language research," in *SOSP '97: Proc. of the 16th ACM symposium on Operating systems principles*. Saint Malo, France: ACM, 1997, pp. 38–51.

[30] A. Reid, M. Flatt, L. Stoller, J. Lepreau, and E. Eide, "Knit: component composition for systems software," in *OSDI'00: Proc. of the 4th Conf. on Symposium on Operating System Design & Implementation*. San Diego, California: USENIX Association, 2000, pp. 24–24.

[31] A. Taherkordi, Q. Le-Trung, R. Rouvoy, and F. Eliassen, "WiSeKit: A distributed middleware to support application-level adaptation in sensor networks," in *DAIS '09: Proc. of the 9th IFIP WG 6.1 Int. Conf. on Distributed Applications and Interoperable Systems*. Lisbon, Portugal: Springer-Verlag, 2009, pp. 44–58.

[32] A. Taherkordi, R. Rouvoy, Q. Le-Trung, and F. Eliassen, "A self-adaptive context processing framework for wireless sensor networks," in *MidSens '08: Proc. of the 3rd Int. Workshop on Middleware for WSNs*. Leuven, Belgium: ACM, 2008, pp. 7–12.

Chapter 11

A Component-based Approach for Service Distribution in Sensor Networks

Authors. Amirhosein Taherkordi (1), Romain Rouvoy (2), and Frank Eliassen (1)

Affiliation.

(1) Department of Informatics, University of Oslo, Norway
{amirhost,frank}@ifi.uio.no

(2) INRIA Lille Nord Europe, ADAM Project-team, University Lille 1, LIFL CNRS UMR 8022, Villeneuve dAscq, France
{romain.rouvoy}@inria.fr

Publication. The 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'10), co-located with ACM/IFIP/USENEX 11th Middleware Conference, Bangalore, India, December 2010.

Abstract. The increasing number of distributed applications over *Wireless Sensor Networks* (WSNs) in ubiquitous environments raises the need for high-level mechanisms to distribute sensor services and integrate them in modern IT systems. Existing work in this area mostly focuses on low-level networking issues, and fails to provide high-level and off-the-shelf programming abstractions for this purpose. In this paper, we therefore consider WSN programming models and service distribution as two interrelated factors and we present a new *component-based* abstraction for integrating WSNs within existing IT systems. Our approach emphasizes on reifying distribution strategies at the software architecture level, thus allowing remote invocation of component services, and facilitating

interoperability of sensor services with the Internet through *Web service-enabled* components. The latter is efficiently provided by incorporating the REST architectural style—emphasizing on abstraction of high-level services as resources—to our component-based framework. The preliminary evaluation results show that the proposed framework has an acceptable memory overhead on a TelosB sensor platform.

11.1 Introduction

WSNs have been recognized as an emerging technology for monitoring and controlling a variety of phenomena, such as environmental surveillance, infrastructures, home and office, and medical environments [1, 2, 3]. Whereas the early WSN applications were primarily concerned with sensing primitive environmental data and delivering those raw data to a central node, recent applications consider sensor nodes as *service-enabled devices* with the ability of providing distributed services to other nodes of the network.

The other concern is related to using WSNs in ubiquitous computing environments, where sensor nodes populate with actuators, RFID readers, and mobile devices for monitoring ambient environments and reacting to the external stimuli gathered by different devices [4, 5, 6]. As each device in such a heterogeneous environment has its own requirements in terms of system software and communication protocol, integrating them at a higher software level brings the challenges of service orchestration and interoperability. Furthermore, sensor nodes as one of the technologies driving the future *Internet of Things* [7] should be equipped with protocols that enable them to interoperate with every IP-enabled node across the Internet.

In contrast to the approach of *plain text message streaming* widely adopted in traditional "sense and send" models [8, 9], the service-oriented approaches in WSNs rely on high-level ad-hoc communication protocols to discover and exhibit services across different computing environments. Although there have been a number of significant efforts to distribute sensor services and integrate them with conventional network platforms [10, 11, 12], the state-of-the-art mostly focuses on low-level APIs and fails to provide a concrete communication abstraction relieving the programmer from dealing with the tedious and error-prone distribution tasks in WSNs.

Thus, in this paper we aim at providing a software framework for WSNs that enables programmers to develop distributed sensor services and integrate them with existing IT systems at a high-level programming abstraction. To achieve that, we leverage the concepts of *component-based programming* to design, describe and implement distributed and interoperable WSN services. In particular, we reconsider Remora—a lightweight component model for high-level programming in WSNs [13]—in order to extend it with the capability of distributing a component's services across the network. The second part of the paper is devoted to integrating the above component-based solution to a uniform interaction model in order to facilitate interoperability of sensor services with the Internet through *Web service-enabled components*. This interaction model is inspired by REST—an archi-

tectural style for distributed systems emphasizing scalability of component interactions, generality of interfaces, and independent deployment of components [14]. From another point of view, this paper illustrates the feasibility of utilizing a component framework to develop RESTful Web services for resource-constrained sensor nodes.

The remainder of this paper is organized as follows. Section 11.2 gives a survey of existing approaches. In Section 11.3, we briefly discuss the REMORA component model. The architecture and specification of our service distribution proposal are presented in Section 11.4, while the implementation and the evaluation result are discussed in Sections 11.5 and 11.6, respectively. Finally, Section 11.7 concludes this paper and identifies some future work.

11.2 Related Work

Efforts in providing distribution and integration approaches for WSNs can be categorized into four groups: *i)* providing low-level APIs to enable distribution within sensor networks for exchanging raw sensed data among nodes, *ii)* high-level component-based techniques to enable remote invocation of sensor services, *iii)* exploiting Web services standards to bridge the gap between sensor nodes and the Internet, and *iv)* protocols allowing sensor nodes to connect seamlessly to other devices in pervasive computing environments.

The first group emphasizes on exposing proprietary interfaces in order to unicast or multicast plain text messages across the network. Most of the works in this field are inspired by the concepts of *message-oriented communication*. TINYOS [15] as the most popular operating system for WSNs provides a number of *Active Message Interfaces* to abstract the underlying radio communications services and *Software Components* that implement these interfaces [8]. Other sensor operating systems, such as CONTIKI [16], have also paved the same way [17]. They have also paid a significant consideration on integrating WSNs with the Internet by providing the IP protocol stack for low-power sensor nodes. However, high-level programming APIs and application-level service distribution facilities are not addressed by these proposals.

A number of lightweight component models have also been proposed to provide RPC-like service invocations in sensor networks. In addition to the fact that none of them consider Web service-based distribution, they essentially suffer from making extensive use of sensor resources or lack of generality. As an example of the former, OPENCOM [18] offers *Component Frameworks* (CFs) to model local and distributed interactions between cooperating components, but OPENCOM is a generic model and basically developed for resource-rich platforms. LOOCI [19], falling in the latter category, is a loosely-coupled component infrastructure for WSNs, featuring an *Event Bus* to bind distributed LOOCI components. Nonetheless, the Java-based implementation of LOOCI limits its usage to the SUNSPOT nodes.

The third group has been motivated by the concept of Internet of Things—a technological revolution to connect daily objects and devices to large databases and networks,

and therefore to the Internet. In this model, Web services standards are used to integrate WSNs and the Internet, *e.g.*, in SOCRADES [20] Web services are tailored at the gateway device where the *Device Profile for Web Services* (DPSW) is used to enable messaging, discovery and eventing on devices with resource restrictions. However, since the current footprint of DPSW for sensor nodes is too large, this solution is only deployable on gateways. To overcome this issue, Priyantha et al. [11] propose a SOAP-based Web services, called *Tiny web services*, for WSNs. However, apart from its complexity, this work mainly focuses on low-level issues related to Web integration in TinyOS-based sensor networks.

A few works have also been devoted to the use of simple Internet protocols. In fact, these approaches are proposed due to the high resource needs and complexity of SOAP-based Web Service protocols for WSNs. TinyREST is one of the first attempts to integrate WSNs into the Internet [21]. It uses the HTTP-based REST architecture to retrieve/update the state of sensors/actuators. The TinyREST gateway maps a set of HTTP requests to TinyOS messages in order to link MICA motes to any Internet client. Beside the fact that in TinyREST only a gateway is able to connect to the Internet (not any individual sensor node), this approach fails to follow all standard HTTP methods. The work reported in [10] also presents a REST-based gateway to bridge the Web requests to powerful SunSPOT nodes.

Approaches in the last category are based on the *Universal Plug and Play* (UPnP) architecture. UPnP is a set of computer network protocols, promoted by the UPnP Forum [22], allowing devices to connect seamlessly in the home and corporate environments. In UPnP, all communications are peer-to-peer and transferred over TCP/IP, UDP and HTTP. As most sensor node products are not equipped with the support of UPnP standards, in [23] a new WSN platform is proposed to support connectivity of sensor nodes to a UPnP-enabled device. However, UPnP discovery and control protocols are heavyweight for a typical sensor node and there is a very limited range of sensor nodes supporting UPnP-based communications. Furthermore, using UPnP in this framework imposes many new hardware and new software set-up for integration support.

The main difference between our approach and the works presented above is that we address the service distribution problem in a top-down manner. Specifically, we first formulate high-level service distribution requirements based on a programming model for WSNs, and then consider how the programming proposal can *simplify* and *generalize* distribution and integration of sensor services.

11.3 Programming Model

We adopt a component-oriented approach to address the programming needs of service distribution in WSNs. Componentization provides a high-level programming abstraction by enforcing interface-based interactions between system modules and therefore avoiding any hidden interaction via direct function call, variable access, or inheritance relationships [24]. This abstraction instead offers the capability of black-box integration of system services.

Therefore, it theoretically becomes a good candidate for developing distribution tasks in WSNs, beside the fact that component-based software development is extensively used in WSN programming. Thus, in this section we briefly discuss a WSN-specific component model we have recently proposed—REMORA [13]. Then, in the next section, we propose our service distribution model based on this component framework.

11.3.1 REMORA in a Nutshell

The main motivation behind proposing REMORA is to facilitate high-level and event-driven programming in WSNs through a component-based abstraction. REMORA achieves this goal by: *i)* deploying components within a lightweight framework executable on every operating system written in the C language, and *ii)* reifying the concept of *event* as a first-class architectural element simplifying the development of event-oriented scenarios. The latter is one of the key features of REMORA since a programming model for embedded systems is expected to support event-driven design. Reducing software development effort is the other objective of REMORA. A REMORA component is composed of two main artifacts: a component *description* and a component *implementation*.

Component Description. REMORA components are described in XML as an extension of the *Service Component Architecture* (SCA) model [25] in order to make WSN applications compliant with the state-of-the-art componentization standards. Based on the SCA Assembly Language, the component description indicates the specifications of the component including *services*, *references*, *interfaces*, *producers*, *consumers*, and *properties* (cf. Figure 11.1). A service can expose a REMORA interface, which is a separate XML document describing the functions provided by the component. A reference can request a REMORA interface, which describes the operations required by the component. Similarly, a producer identifies an event type generated by the component, while consumer specifies a component's interest on receiving a particular type of event.

Component Implementation. The component implementation contains operations implementing: *i)* the component's service interfaces, *ii)* event handlers, and *iii)* private utilities of the component. REMORA components are implemented by using the C programming language extended with a set of new commands. This extension is essentially proposed to support the main features of REMORA, namely, component instantiation, event processing, and property manipulation.

Remora Development Process. A REMORA application consists of a set of REMORA components, containing descriptions and implementations of software modules (cf. Figure 11.2). The REMORA engine processes the component descriptions and generates standard C code deployable within the REMORA framework. The framework is an OS-independent C module supporting the specification of the REMORA component model. Finally, the REMORA framework is deployed on the target sensor node through the REMORA runtime, which is an OS-abstraction layer integrating the application with the system software.

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="COMPONENT_NAME">
  <service name="SERVICE1_NAME">
    <interface.remora name="INTERFACE1_NAME"/>
  </service> ...
  <reference name="REFERENCE1_NAME">
    <interface.remora name="INTERFACE2_NAME"/>
  </reference> ...
  <property name="PROP1_NAME" type="PROP1_TYPE">
    PROP1_DEFAULT_VALUE
  </property> ...
  <producer>
    <event.remora type="EVENT1_TYPE"
                  name="EVENT1_NAME"/>
  </producer> ...
  <consumer operation="CONSUMER_OPERATION">
    <event.remora type="EVENT2_TYPE"
                  name="EVENT2_NAME"/>
  </consumer> ...
</componentType>
```

Figure 11.1: The XML template for describing Remora components.

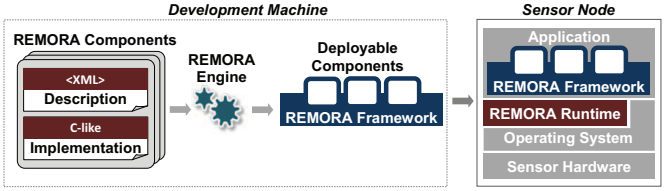

Figure 11.2: Development process of Remora applications.

11.4 Component-Based Service Distribution

Remora provides a complete component-based abstraction for *node-level programming* in the sense that the accessibility scope of Remora services is limited to the runtime process of a sensor node. Services deployed on two different nodes can communicate through an indirect ad-hoc interaction model, which is message-based and handled by low-level data transmission protocols provided by sensor operating systems. It means that for network-level programming, the developer requires to switch from component-based programming to system-level messaging techniques, a cumbersome and error-prone task. It is also the case when a Remora-based application is integrated with heterogeneous systems with a different set of software and hardware settings.

11.4.1 Basic Concepts

We propose a new key concept, which is considered as a basis for our distribution proposal. This concept, called REMORA *Distribution*, refers to a new paradigm in component-based software design for resource-constrained networks in which every system functionality is encapsulated in a component and distributed for the use of both homogenous and heterogeneous systems. This indicates the capability of defining platform-specific *bindings* for REMORA components in order to transparently handle the communication issues in WSNs. In this way, REMORA will automatically generate part of the code which is required for sending data over the network or invoking methods, instead of time-consuming and error-prone programming with low-level APIs.

Generally, the REMORA bindings are categorized into two classes: *remote binding* and *interoperable binding*. The former specifies a type of remote service call occurring within a sensor network between two homogenous sensor nodes in order to send the required service data from one node to the other node (like Java RMI), while the latter refers to the invocations happening beyond a sensor network between a sensor node and a node in an existing IT system, such as the Internet. To concretely describe the REMORA distribution, we first discuss the principles underpinning this framework, including:

Access Transparency. From the programmer's point of view, local and distributed services should be accessed using identical operations. Additionally, she/he should not be forced to write distribution-related code that is out of scope of application logic.

Location Transparency. The distribution mechanism should enable REMORA components to access remote services without any knowledge of their location in the network.

Synchronisms. The calling mode can be either *synchronous* or *asynchronous* corresponding to the mode supported by the network protocol, *e.g.*, RESTful calls are synchronous, while remote calls over RIME—a WSN-specific communication protocol implemented by CONTIKI [17]—are handled asynchronously.

Pluggability of Bindings. REMORA bindings should be maintained in a well-structured way in the sense that every binding-related library for a particular network protocol should be easily pluggable to or unpluggable from services provided by REMORA components without affecting other parts of the system.

Figure 11.3 depicts the overall architecture of the REMORA distribution model. Each component of the system exhibits a set of services, which can interact in two different ways: locally or remotely. In the local mode, service calls are carried out through a simple and lightweight node-level invocation plan, while the distributed calls occur either remotely (between two nodes with the same set of configurations), or in an interoperable manner (between a REMORA-enabled node and an IP-enabled node).

While the development of each main type of REMORA binding would require consideration of many kinds of challenges, such as programming constructs, streaming, networking, and runtime support, the contribution of this paper is limited to providing service-level integration of sensor nodes with IP-enabled systems. We therefore leave the issues of the

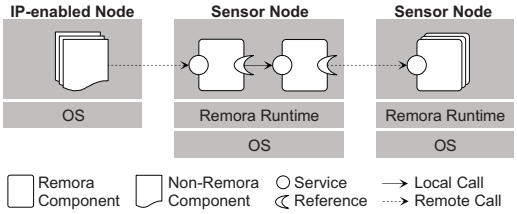

Figure 11.3: Architecture of the Remora distribution model.

Remora remote binding as our future work and in this paper we study the Remora interoperable binding in the context of Web integration. Remora *Web binding* is basically concerned with integrating sensor services with IP-enabled networks by translating the service requests to the equivalent RESTful web services. In this way, the distribution mechanism of Remora follows a standard and widely-adopted protocol to link WSN services to any Web-enabled device in ubiquitous environments.

11.4.2 REMORA Web Services

In this section, we first describe the principles of the REST architectural style and RESTful Web services, and then present our component-based approach for integrating Remora services with the Web.

REST Principles. The *REpresentational State Transfer* (REST), as coined by Fielding [14], is an architectural style for distributed systems emphasizing scalability of component interactions, generality of interfaces, and independent deployment of components. The "resource-oriented" principles of REST are described through the REST triangle defining the principles for *addressing*, *accessing* and *encoding* a collection of resources using the Internet standards. A distributed system, which follows REST principles, is called RESTful, *e.g.*, the Web. In a RESTful system, a component can interact with other distributed components by knowing two things: *i)* the unique identifier of the representative resource of component, and *ii)* the predefined standard operations to invoke (GET, POST, PUT and DELETE). In this model of interaction, the client-server separation of concerns can simplify component implementation, reduce the complexity of connector semantics, and increase the scalability of server components.

RESTful Web services. Web services are a set of standards and techniques for developing interoperable distributed applications that are accessed via HTTP and executed on a remote system hosting the requested services. Nowadays, Web services are generally categorized into two groups: *SOAP-based Web services* and *RESTful Web services*. The former follows the *Simple Object Access Protocol* (SOAP), which is a heavyweight protocol specification for exchanging structured information in the implementation of Web services, while the latter is a lightweight model based on REST, which does not impose SOAP or

XML.

We believe that the simplicity and uniform interfaces of RESTful Web services is an efficient approach to integrate any individual sensor node to any RESTful system in pervasive environments. Additionally, there have been reported a number of valuable works focusing on the integration of REST and WSNs and providing the *primitives* required to RESTful programming in sensor systems [12]. Therefore, we adopt this approach and study how to enable RESTful Web service development in WSNs by the Remora component model.

Since an application built from REST principles is transformed from operation-centric into a data-centric model, every entity that offers a service becomes a resource (*e.g.*, a temperature sensor) that can be identified unambiguously using a Uniform Resource Identifier (URI) [26]. Every resource then defines a uniform interface just including four main operations provided by REST (GET, POST, PUT and DELETE). Therefore, incorporating REST principles into Remora requires applying the REST triangle of nouns, verbs, and content types to the specification of Remora component model.

Remora Service Identifier. The service identifier is a unique noun described using the URI format. Therefore, service identifiers include a server address, a service path, and a sequence of request parameters:

```
/server-address/service-path?request-params
```

Since Remora components in a sensor application are basically organized in a hierarchical model (like Java packages), the URI of a distributed service is corresponding to the hierarchical organization of Remora components within the application. For instance, assume that Sensors is a component providing a service for temperature sensor (`myTemperature`) and another service for light sensor (`myLight`), thus the services of the Sensors component, located under `/app/peripheral`, can be addressed as follow:

```
/node-id/app/peripheral/Sensors/myTemperature
/node-id/app/peripheral/Sensors/myLight
```

As the Remora engine has a complete knowledge about the structure of applications (including hierarchical organization of system components), service identifiers are automatically obtained by the Remora engine and therefore the programmer does not need to specify any identifier for distributed services during the application development. Nevertheless, we have provided a special REST binding tag (`binding.rest`) through which the programmer can change the URI of a component, as well as its services. Figure 11.4 shows the description of the Sensors component, where the URI of the `myTemperature` service is changed. Therefore, the new URI of the `myTemperature` service becomes:

```
/node-id/app/myTemp
```

Remora Service API. The four main operations of REST can be mapped to the operations implemented by a particular Remora service. Figure 11.5 shows an excerpt of the

```
<?xml version="1.0" encoding="UTF-8"?>
<componentType name="app.peripheral.Sensors">
  <service name="myTemperature">
    <interface.remora name="api.ITemperature"/>
    <binding.rest uri="/app/myTemp"/>
  </service>
  <service name="myLight">
    <interface.remora name="api.ILight"/>
  </service>
  ...
</componentType>
```

Figure 11.4: RESTful service identification in Remora.

interface `ITemperature`. Similar to the REMORA services, operations of an interface can be considered as a REST resource and have their own URI. Depending on the functionality of a service and variety of its operations, we may need to define a new URI for a collection of service operations. The other important part of the REST binding tag is that we need to provide the name of the equivalent REST operation for each operation. For example, the current temperature of environment can be retrieved from a temperature sensor node using the HTTP request:

```
GET http://device.uio.no:8080/node-id/app/myTemp
```

which returns the current temperature. Temperature configuration information can also be pushed into a sensor node by using an HTTP PUT request as follow:

```
PUT http://device.uio.no:8080/node-id/app/
                                  myTemp/threshold
```

In this case, the HTTP request is sent to the operation `setThreshold`, along with the required parameters, then the REMORA framework will call the `setThreshold` operation. Note that in the sample code of Figure 5, `threshold` is considered as a new separate REST resource with its own URI.

Content Type. Finally, for a REST binding we can specify the format of data delivered to the client. For instance, according to the REST annotation defined for `getCurrent` in the `ITemperature` interface, this operation provides the current temperature in the JSON format (cf. Figure 11.5).

11.4.3 A Concrete Use Case

An example use case that can benefit greatly from our distribution framework is *home monitoring systems*. Such applications are characterized as being filled with sensor nodes to observe various types of ambient context elements (temperature, smoke, occupancy, and health conditions of inhabitant), actuators to physically control home appliances (lights,

```
<?xml version="1.0" encoding="UTF-8"?>
<interface.remora name="api.ITemperature">
  <operation name="getCurrent" return="xsd:short">
    <annotation.rest method="GET"
                     type="application/json"/>
  </operation>
  <operation name="setThreshold" return="void">
    <in name="threshold" type="xsd:short"/>
    <annotation.rest method="PUT" path="/threshold"/>
  </operation>
  ...
</interface.remora>
```

Figure 11.5: Mapping the REST verbs to equivalent Remora operations and identifying content types.

TV, and air conditioning), and smart phones to provide information about the preferences of owners.

Integration of multi-scale entities is one of the main challenges in such an environment. Mobile devices and sensors have different hardware and software capabilities, which make some devices more powerful than others. Therefore, this heterogeneity requires a flexible and simple solution that supports multiple interaction mechanisms and considers the restricted capabilities of some devices. In particular, regarding sensor nodes, the immaturity of high-level communication protocols, as well as the inherent resource scarceness, bring a critical challenge to the system: how to connect sensor nodes to mobile devices and actuators through a standard high-level communication protocol. We believe that the RESTful framework presented above can significantly smooth the way to integrate a sensor network with actuators and existing infrastructure networks in the home monitoring systems.

11.5 Implementation

The implementation of the proposed model consists of two main parts according to the model's architecture. The first part is concerned with modifying the implementation of Remora specifications in order to support the new commands related to Web services programming, such as the REST binding tag. The second part is dedicated to developing a middleware framework taking care of REST communication issues. The former is performed within the implementation of the Remora engine. In addition to enhancing the engine to support the new commands, it should statically create the data structures required for maintaining the REST-related service data, such as resource identifiers and Service API information.

The middleware framework, supporting Web service-based communications, is a lightweight module integrated with the Remora runtime and the Remora framework to support runtime requirements of the proposed model. To this end, we first need to choose and exploit an existing library that is flexible and completely support the REST principles. We adopt

the *REST framework* presented by Yazar et al. [12] for two reasons. Firstly, they have used the Contiki operating system to develop the RESTful architecture and our current Remora runtime is also available on Contiki. Secondly, this framework's architecture along with Contiki's extensive work on the IP protocol stack provide a set of complete and easy-to-use low-level libraries for RESTful enhancements.

Figure 11.6 depicts the overall implementation architecture of Remora Web services, where Contiki's core and TCP/IP stack lie in the bottom. The gray box represents the REST framework and contains core modules required for RESTful Web services development over Contiki. The Remora runtime is integrated with the REST framework through the *REST Wrapper API*. It should be noted that Wrapper API is one of the main features of Remora, providing a well-described method for integrating a Remora application with underlying system software [13].

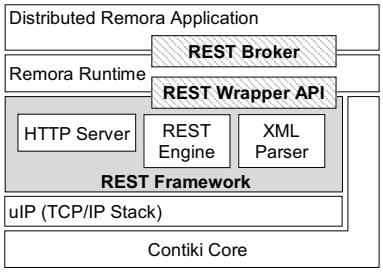

Figure 11.6: Overall implementation architecture of Remora Web services.

Figure 11.7 shows an excerpt of the REST Wrapper API, including operations for initializing REST engine, setting the representation type for a HTTP connection through the Remora component HTTPConnection, sending GET data, receiving POST data, and updating the status of a connection. The implementer of this interface is a Remora component in which the low-level APIs of the REST framework are invoked.

REST Broker contains a set of Remora components processing REST requests received from Remora runtime. Specifically, it is an intermediate module for handling the REST requests received from a Web client or sent from the sensor node to a node hosting RESTful Web services. The broker is also in charge of retaining the list of application-specific resources and the corresponding Remora Web services APIs.

11.6 Preliminary Evaluation

As mentioned before, we adopt Contiki as our system platform and our hardware platform is the popular TelosB mote equipped with a 16-bit TI MSP430 MCU with 48KB ROM and 10KB RAM. In the preliminary evaluation, we assess the performance of the system

```
<?xml version="1.0" encoding="UTF-8"?>
<interface.remora name="api.wrapper.IREST">
  <operation name="initialize"/>
  <operation name="setReperType">
    <in name="httpConnection"
        type="rest.comm.HTTPConnection"/>
    <in name="type" type="xsd:int"/>
  </operation>
  <operation name="setGETData">
    <in name="httpConnection"
        type="rest.comm.HTTPConnection"/>
    <in name="data" type="xsd:string"/>
  </operation>
  <operation name="getPOSTData" return="xsd:string"/>
  <operation name="setHTTPStatus">
    <in name="httpConnection"
        type="rest.comm.HTTPConnection"/>
    <in name="type" type="xsd:int"/>
  </operation>
  ...
</interface.remora>
```

Figure 11.7: An excerpt of the REST Wrapper API.

based on the memory overhead incurred by the REST Broker and the REST wrapping component (implementer of the REST Wrapper API) on ROM and RAM.

The memory footprints are categorized into a fixed overhead and a dynamic overhead. The fixed overhead is the minimum additional memory required for a distributed application, regardless of the number of distributed services running within the application. Table 11.1 shows the fixed memory requirements, which turn out to be quite reasonable with respect to both code and data memory. As seen from the table, the main memory consuming module is the REST framework. Therefore, the fixed memory cost can be further reduced by switching to a more efficient REST framework in the future.

Table 11.1: The fixed memory requirement of Remora Web services framework.

Module	**Code Memory (bytes)**	**Data Memory (bytes)**
Remora Runtime	494	14
REST Wrapper	94	0
REST Broker	252	8
REST Framework	4668	76
Total	**5508**	**98**

The dynamic memory overhead is calculated based on the number of distributed resources (services, components or operations) in an application and the number of REST verb mappings (GET, POST, PUT and DELETE) for each resource. The memory over-

head of the former is variable according to the length of a resource's name, while the latter consumes 2 bytes of ROM to retain the starting memory address of a REST operation.

11.7 Conclusions and Future Work

Distributing WSN services through standard and widely-accepted communication protocols is of high importance. In this paper, we presented a high-level programming abstraction in order to enable service distribution in WSN applications and relieve the programmer from the burden of dealing with low-level APIs for developing distributed sensor services. This component-based approach promises a new abstraction for integrating sensor software modules to the Internet through upgrading component-level services to Web services over a lightweight RESTful architecture. This flexible framework is also potentially able to exhibit sensor services to other types of network protocols by implementing platform-specific bindings. Our future work includes addressing the REMORA remote binding, occurring when component services within a sensor network need to remotely communicate via a particular sensor network protocol.

References

[1] M. Ceriotti *et al.*, "Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment," in *IPSN '09: Proc. of the 2009 Int. Conf. on Information Processing in Sensor Networks*. San Francisco, CA, USA: IEEE, 2009, pp. 277–288.

[2] A. Milenković, C. Otto, and E. Jovanov, "Wireless sensor networks for personal health monitoring: Issues and an implementation," *Computer Communications (Special issue: Wireless Sensor Networks: Performance, Reliability, Security, and Beyond)*, vol. 29, pp. 2521–2533, 2006.

[3] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh, "Monitoring volcanic eruptions with a wireless sensor network," in *EWSN '05: Proc. of the Second European Workshop on Wireless Sensor Networks*, Istanbul, Turkey, 2005, pp. 108–120.

[4] I. F. Akyildiz and I. H. Kasimoglu, "Wireless sensor and actor networks: research challenges," *Ad Hoc Networks*, vol. 2, no. 4, pp. 351–367, 2004.

[5] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the physical world with pervasive networks," *IEEE Pervasive Computing*, vol. 1, no. 1, pp. 59–69, 2002.

[6] H. Liu, M. Bolic, A. Nayak, and I. Stojmenovic, "Taxonomy and Challenges of the Integration of RFID and Wireless Sensor Networks," *IEEE Network*, vol. 22, no. 6, pp. 26–35, 2008.

[7] N. Gershenfeld, Raffi, R. Krikorian, and D. Cohen, "The internet of things," *SCIENTIFIC AMERICAN*, pp. 76–81, 2004.

[8] P. Buonadonna, J. Hill, and D. Culler, "Active message communication for tiny networked sensors," in *INFOCOM '01: Proc. of the 20th Annual Joint Conf. of the IEEE Computer and Communications Societies*. Alaska, USA: IEEE, 2001.

[9] E. Souto, G. Guimar aes, G. Vasconcelos, M. Vieira, N. Rosa, and C. Ferraz, "A message-oriented middleware for sensor networks," in *MPAC '04: Proc. of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*. Toronto, Canada: ACM, 2004, pp. 127–134.

[10] D. Guinard, V. Trifa, T. Pham, and O. Liechti, "Towards physical mashups in the web of things," in *INSS '09: Proc. of the 6th Int. Conf. on Networked Sensing Systems*. Pittsburgh, PA, USA: IEEE, 2009, pp. 196–199.

[11] N. B. Priyantha, A. Kansal, M. Goraczko, and F. Zhao, "Tiny web services: Design and implementation of interoperable and evolvable sensor networks," in *SenSys '08: Proc. of the 6th ACM Conf. on Embedded Network Sensor Systems*. Raleigh, NC, USA: ACM, 2008, pp. 253–266.

[12] D. Yazar and A. Dunkels, "Efficient application integration in ip-based sensor networks," in *BuildSys '09: Proc. of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings*. Berkeley, CA, USA: ACM, 2009, pp. 43–48.

[13] University of Oslo, "The REMORA Component Model," 2010, http://folk.uio.no/amirhost/remora.

[14] R. Fielding, *Architectural Styles and the Design of Network-based Software Architectures*, University of California, Irvine, USA, 2000, PhD thesis.

[15] P. Levis, S. Madden, J. Polastre, R. Szewczyk, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler, "Tinyos: An operating system for sensor networks," in *Ambient Intelligence*. Berlin, Germany: Springer Verlag, 2004, pp. 15–148.

[16] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *LCN '04: Proc. of the 29th Annual IEEE Int. Conf. on Local Computer Networks*. Tampa, Florida, USA: IEEE Computer Society, 2004, pp. 455–462.

[17] A. Dunkels, F. Österlind, and Z. He, "An adaptive communication architecture for wireless sensor networks," in *SenSys '07: Proc. of the 5th Int. Conf. on Embedded networked sensor systems*. Sydney, Australia: ACM, 2007, pp. 335–349.

[18] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan, "A generic component model for building systems software," *ACM Trans. Comput. Syst.*, vol. 26, no. 1, pp. 1–42, 2008.

[19] D. Hughes, K. Thoelen, W. Horré, N. Matthys, P. J. del Cid Garcia, S. Michiels, C. Huygens, and W. Joosen, "Looci: A loosely-coupled component infrastructure for networked embedded systems," in *Proc. of the 7th Int. Conf. on Advances in Mobile Computing & Multimedia*. Kuala Lumpur, Malaysia: ACM, Dec. 2009, pp. 195–203.

[20] L. de Souza, P. Spiess, D. Guinard, M. Khler, S. Karnouskos, and D. Savio, "Socrades: A web service based shop floor integration infrastructure," in *The Internet of Things*, ser. LNCS, vol. 4952. Springer, 2008, pp. 50–67.

[21] T. Luckenbach, P. Gober, K. Kotsopoulos, Andreas Kim, and S. Arbanowski, "Tinyrest: a protocol for integrating sensor networks into the internet," in *REALWSN '05: Proc. of the Workshop on Real-World WSNs*, Stockholm, Sweden, 2005.

[22] UPnP Forum, "UPnP Device Architecture 1.0," http://www.upnp.org/resources/documents.asp, Apr. 2008.

[23] M. Marin-Perianu *et al.*, "Decentralized enterprise systems: a multi-platform wireless sensor network approach," *Wireless Communications, IEEE*, vol. 14, no. 6, pp. 57–66, 2007.

[24] C. Szyperski, *Component Software: Beyond Object-Oriented Programming, 2nd edition*. Boston, MA, USA: Addison-Wesley, 2002.

[25] OSOA, "The service component architecture," http://www.oasis-opencsa.org/sca.

[26] R. T. Berners-Lee and M. Fielding, L., "Uniform Resource Identifier (URI): Generic Syntax," 2005, http://www.ietf.org/rfc/rfc3986.txt.

Chapter 12

The DigiHome Service-Oriented Platform

Authors. Daniel Romero (1), Gabriel Hermosillo (1), Amirhosein Taherkordi (2), Russel Nzekwa (1), Romain Rouvoy (1), and Frank Eliassen (2)

Affiliation.

(1) INRIA Lille Nord Europe, ADAM Project-team, University Lille 1, LIFL CNRS UMR 8022, Villeneuve dAscq, France
{firstname.lastname}@inria.fr

(2) Department of Informatics, University of Oslo, Norway
{amirhost,frank}@ifi.uio.no

Publication. Accepted to Software: Practice & Experience Journal. The earlier version of this paper has been published in the proceedings of the 10th IFIP International Conference on Distributed Applications and Interoperable Systems (DAIS), Amsterdam, Netherlands, June 2010.

Abstract. Nowadays, the computational devices are everywhere. In malls, offices, streets, cars and even in homes we can find devices providing and consuming functionality in order to improve the user satisfaction. These devices include sensors that provide information about the environment state (*e.g.*, temperature, occupancy, light levels), service providers (*e.g.*, Internet TVs, GPS), smartphones (that contain user preferences), and actuators that act on the environment (*e.g.*, closing the blinds, activating the alarm, changing the temperature). Although these devices exhibit communication capabilities, their integration into a larger monitoring system remains a challenging task, partly due to the strong

heterogeneity of technologies and protocols. Therefore, in this article we focus on home environments and propose a middleware solution, called DigiHome, which applies the SCA (*Service Component Architecture*) component model in order to integrate data and events generated by heterogeneous devices in this kind of environments. DigiHome exploits the SCA extensibility to incorporate the REST (*REpresentational State Transfer*) architectural style, and in this way leverages on the integration of multi-scale systems-of-systems (from Wireless Sensor Networks to the Internet). Additionally, the platform applies CEP (*Complex Event Processing*) technology that detects application-specific situations. We claim that the modularization of concerns fostered by DigiHome and materialized in a service-oriented architecture, makes it easier to incorporate new services and devices in smart home environments. The benefits of the DigiHome platform are demonstrated on smart home scenarios covering home automation, emergency detection, and energy saving situations.

12.1 Introduction

Pervasive environments support context-aware applications that adapt their behavior by reasoning dynamically about the user and the surrounding information. This contextual information generally comes from diverse and heterogeneous entities, such as physical devices, *Wireless Sensors Networks* (WSNs), and smartphones. In order to exploit the information provided by these entities, a middleware solution is required to collect, process, and distribute the contextual information efficiently. However, the heterogeneity of systems in terms of technology capabilities and communication protocols, the mobility of the different interacting entities, and the identification of adaptation situations make this integration difficult. Thus, this challenge requires a flexible solution in terms of communication support and context processing to leverage context-aware applications on the integration of heterogeneous context providers.

In particular, a solution dealing with context information and control environments must be able to connect with a wide range of device types. However, the resource scarcity in WSNs and mobile devices makes the development of such a solution very challenging. In this article, we propose the DigiHome platform, an improved version of our work introduced in [1]. With this platform we provide a simple but efficient service-oriented middleware solution to facilitate context-awareness in pervasive environments. Specifically, DigiHome supports the *integration*, *processing* and *adaptation* of the context-aware applications. Our solution enables the integration of heterogeneous computational entities by relying on the Service Component Architecture (SCA) model [2], the REST (*REpresentational State Transfer*) principles [3], standard discovery and communication protocols, and resource representation formats. We combined SCA and REST in our solution in order to foster reuse and loose coupling between the different services that compose the platform. Furthermore, while our solution also benefits from WSNs to operate simple event reasoning on the sensor nodes, we rely on *Complex Event Processing* [4] for analyzing in real-time

the relationships between the different collected events and trigger rule-based adaptations.

The remainder of this article is organized as follows. We start by describing a smart home scenario in which we identify the key challenges in pervasive environments that motivate this work (cf. Section 12.2). Then, we present some of the background concepts that we use in our project (cf. Section 12.3). We continue by the description of DigiHome, our middleware platform to support the integration of systems-of-systems in pervasive environments (cf. Section 12.4). Then, we discuss the benefits of our approach (cf. Section 12.5) before presenting the related work (cf. Section 12.6). Finally, we conclude by presenting some promising perspectives for this work (cf. Section 12.7).

12.2 Motivating Scenario

In this paper, we use a smart home scenario to show the motivation of our work. A smart home generally refers to a house environment equipped with several types of computing entities, such as *sensors*, which collect physical information (temperature, movement detection, noise level, light, etc.), and *actuators*, which change the state of the environment. Sensor nodes are mostly embedded in home appliances and may be powered by batteries with limited capacity. In this scenario, we consider a smart home equipped with occupancy, smoke detection, and temperature sensors. These tiny devices have the ability to collect context information and to communicate wirelessly with each other, in order to identify the context situation of the environment. In addition to that, we can also use actuators to physically control lights, TV, and air conditioning. Figure 12.1 illustrates the integration of these sensors and actuators in our scenario. As observed in this figure, the different entities use heterogeneous protocols to interact. In the scenario, the smartphones provide information about the user preferences for the home configuration. When several people share the same room, the configuration decision is based on merged preferences. Conflicts between the user preferences are resolved by giving, *e.g.*, priority to the person who arrived first to the room. The mobile devices also have an application that enables the control of the actuators present in the different rooms. This application can be adapted when there are changes in the actuator's configuration. Finally, there is a *Controller* device, which is able to gather information, and interact with the other co-located devices.

To show how the different elements of our scenario interact, we present three different situations:

Situation 1: Alice arrives to the living room. The occupancy sensor detects her presence and triggers the temperature sensors to increase the sampling rate of data. It also notifies the Controller that the room is occupied by somebody, which in turn tries to identify the occupant by looking for a profile in her mobile device. When Alice's profile is found, the Controller loads it and adjusts the temperature and lightening level of the room according to Alice's preferences.

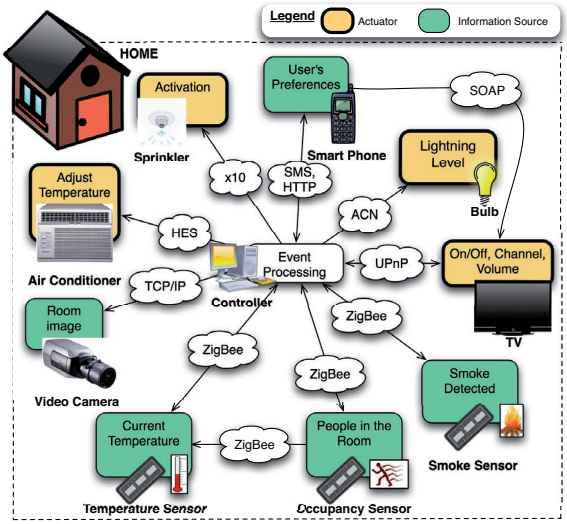

Figure 12.1: Interactions between the smart home devices.

Situation 2: The sensors detect smoke and notify the Controller, which using the occupancy sensor, detects that the house is empty. The Controller therefore sends an SMS to Alice, including a picture of the room captured using the surveillance camera. After checking the picture, Alice decides to remotely trigger the sprinklers using her mobile device. She also tells the system to alert the fire department about the problem. If Alice does not reply to the Controller within 5 minutes, the system activates automatically the sprinklers and alerts the fire department.

Situation 3: Alice installs a new TV in the bedroom. The Controller detects the presence of the new device, identifies it, and downloads the corresponding control software from an Internet repository. The platform tries to locate the available mobile devices, using a discovery protocol, and finds Alice's mobile device. The Controller proposes to update the mobile device with the components for controlling the new TV.

12.2.1 Key Challenges

The various situations we described above allow us to identify several key challenges in terms of:

1. *Integration of multi-scale entities*: The mobile devices and sensors have different

hardware and software capabilities, which make some devices more powerful than others. Therefore, the integration of these entities requires a flexible and simple solution that supports multiple interaction mechanisms and considers the restricted capabilities of some devices. In particular, regarding sensor nodes, the immaturity of high-level communication protocols, as well as the inherent resource scarcity, bring two critical challenges to our work: *1)* how sensor nodes should be connected to mobile devices and actuators through a standard high-level communication protocol, and *2)* the framework which runs over sensor nodes for supporting context-awareness and adaptation should not impose high resource demands.

2. *Entities mobility*: In our scenario, computational entities appear and disappear constantly. In particular, mobile devices providing user profiles are not always accessible (they can be turned off or the owner can leave the house with them). In a similar way, the actuators can be replaced or new ones can be added. Thus, we need to discover new entities dynamically as well as to support device disconnections.

3. *Information processing and adaptation*: In order to support adaptation, we first need to identify the situations, in which the adaptation is required. We have a lot of information that is generated by the different devices in the environment. Therefore, we need to define which part of this information is useful to identify relevant situations and react accordingly. In our scenario, those situations include the load of Alice's profile and the adjustment of the temperature, the sending of alerts via SMS in case of an emergency, and the adaptation of Alice's mobile device to control the new TV in her bedroom.

12.3 Background

In this section we present a brief introduction to the main elements employed in the DigiHome conception: The Service Component Architecture Model (SCA), the FraSCAti platform and Complex Event Processing technology.

12.3.1 Service Component Architecture (SCA) Model

SCA is a set of specifications for building distributed applications based on *Service-oriented architecture* (SOA) and *Component-Based Software Engineering* (CBSE) principles [2]. In SCA, the basic construction blocks are the *software components*, which have *services* (or provided interfaces), *references* (or required interfaces) and exposed properties. The references and services are connected by means of *wires*. SCA specifies a hierarchical component model, which means that components can be implemented either by primitive language entities or by subcomponents. In the latter case the components are called *composites*.

SCA is designed to be independent from programming languages, *Interface Definition Languages* (IDL), communication protocols and non-functional properties. In this way, an SCA-based application can be built, for example, using components in Java, PHP, and COBOL. Furthermore, several IDLs are supported, such as WSDL and Java Interfaces. In order to support interaction via different communication protocols, SCA provides the concept of *binding*. For SCA references, *bindings* describe the access mechanism used to call a service. In the case of services, the bindings describe the access mechanism that clients have to use to call the service.

12.3.2 The FRASCATI platform

The FraSCAti platform [5, 6] allows the development and execution of SCA-based applications. The platform itself is built as an SCA application—*i.e.*, its different subsystems are implemented as SCA components. FraSCAti provides an homogeneous view of a middleware software stack where the platform, the non-functional services, and the applications are uniformly designed and implemented with the same component-based and service-oriented paradigm. To achieve this, the platform is composed of four layers: *i)* the *Kernel Level* based on Fractal [7], a lightweight and open component framework with basic dependency injection, introspection and reconfiguration capabilities; *ii)* the *Personality Level*, which customizes the component kernel by providing the components with execution semantics and implementing the SCA API and principles based on the Fractal component model; *iii)* the *Runtime Level* that instantiates SCA assemblies and components and defines a flexible configuration process, which is inspired by the *extender* and *whiteboard* [8] design patterns of OSGi; and *iv)* the *Non-Functional Level* that supports the SCA Policy Framework specification in order to provide non-functional services implemented as regular SCA components.

12.3.3 Complex Event Processing

CEP is an emerging technology for finding relationships between series of simple and independent events from different sources, using previously defined rules [4]. It employs different techniques such as detection of complex patterns, event correlation and abstraction, event hierarchies and relationships between events using causality, membership, and timing. It is used in a variety of domains such as logistics, transport and finance. In our scenario, we consider a lot of heterogeneous devices (sensors, mobile devices, etc.) that generate isolated events, which can be used to obtain valuable information and to make decisions accordingly.

To understand the concept, let us consider the examples in the scenario of Section 12.2. For instance, CEP can be used for simple events, like detecting the presence of a person in the house and triggering the discovery service to identify that person. However, whenever a person moves in the room, the presence event will be received. In order to prevent the triggering of the discovery service every time a person moves in the room, with CEP we

can use windows of time. Using them, we can specify that we are only interested in those events every number of seconds or minutes.

Moreover, CEP can be used to find relationships between isolated events. For example, if the smoke detectors send an event and the temperature is above 40ºC, then we can assume that there is fire in the room and alert the user. Using again the windows of time, we could specify that if within 5 minutes the user has not responded to the alert, then it should trigger the sprinklers and send an alert to the fire department.

Finally, using CEP we can also configure some comfort rules according to user preferences. For example, if the user turns on the TV in a room with a window, and there is too much light outside, then it could close the blinds to improve the user's experience.

12.4 The DigiHome Service-Oriented Platform

The integration, mobility and adaptation issues impose several requirements for the development of smart homes environments. To deal with these issues, in this section we propose a comprehensive and simple solution called DigiHome, which enables the integration of events and context information as well as the dynamic configuration of applications. In particular, we propose a flexible architecture that modularizes the different concerns associated with event processing in ubiquitous environments by applying existing standards and approaches. In our solution, we support the integration of events sources (*e.g.*, sensors in our scenario), context providers (*e.g.*, mobile devices) and other kind of services (*e.g.*, actuators and reconfiguration services) implemented with a variety of technologies and interacting via different protocols by means of the SCA component model. Indeed, DigiHome deals with protocol heterogeneity, by enabling the incorporation at runtime of different communication mechanisms if required thanks to the SCA feature isolation of non-functional concerns.

In DigiHome, we follow the REST principles [3] to reduce the coupling between entities by focusing the interaction in the exchanged data, which can have multiple representations (*e.g.*, XML and JSON). In a similar way, for supporting the integration of devices with restricted capabilities, DigiHome promotes the usage of a lightweight API and simple communication protocols as stated by REST. In particular, our solution benefits from WSNs in order to process simple events and make local decisions when possible, by means of the Remora component model [9], which is a component model for WSNs based on SCA. Finally, the platform uses a CEP engine for the adaptation of applications and room configuration. Figure 12.2 depicts the general architecture of the platform. In the rest of the section we provide a detailed description of the different elements of the platform.

12.4.1 DigiHome Core

The core of the platform modularizes the main responsibilities for home monitoring. This means that the core contains the functionality required for event collecting, event process-

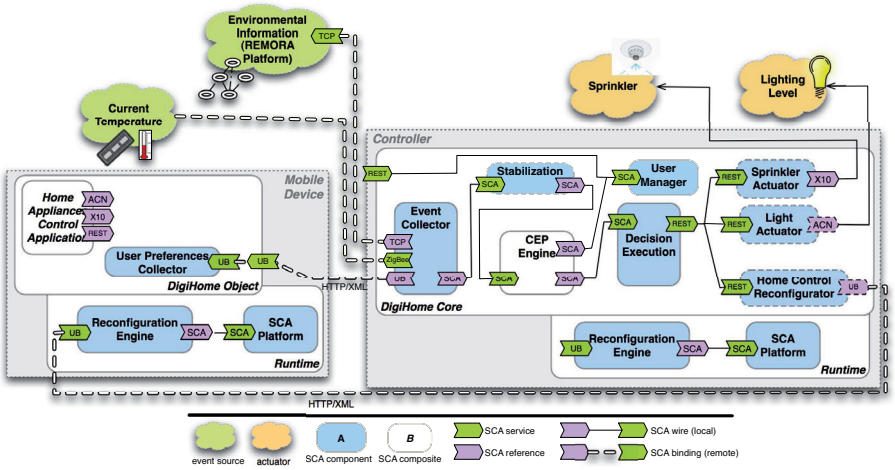

Figure 12.2: Description of the DigiHome architecture.

ing, and deciding and executing the required adaptations of the applications deployed on DigiHome objects (cf. Section 12.4.2) as well as the room configurations. In DigiHome, the Event Collector retrieves and stores the recent information produced by event and context sources, such as sensors and mobile devices. The CEP Engine is responsible for event processing and uses the Decision Executor to perform actions specified by the Adaptation Rules (defined in the CEP Engine). The CEP Engine also employes a User Manager service for determining if inhabitants have rights for executing specific system operations (*e.g.*, the activation of sprinklers or the alert of the fire department in situation 2) and who must to be notified in case of requiring human intervention. Furthermore the User Manager service is used by the Event Collector for deciding if the information from a mobile device has to be processed by the system or not.

On the other hand, the core contains different Actuator components that grant access to the available actuator services in the environment. Following a plug-in mechanism, these components can be installed or uninstalled at runtime. This means that the different actuators are optional, deployed according to the current service configuration and installed on different devices.

To enable the communication between different clients and to support the mobility of services and mobile devices, we incorporate ubiquitous bindings in SCA [10]. These bindings bring into SCA existing discovery protocols, such as UPnP [11] and SLP [12], providing the possibility to establish spontaneous communication. Furthermore, the ubiquitous bindings improve the context information advertisements with *Quality of Context*

(QoC) [13] attributes for provider selection. Once the services are discovered, the ubiquitous bindings are flexible enough to allow the interaction via standard bindings, such as REST. The use of these ubiquitous bindings, as well as the modularization of the different concerns, makes it easy to distribute the different responsibilities in DigiHome.

12.4.2 DigiHome Objects

A DigiHome Object is an SCA component providing and/or consuming events to/from other DigiHome Objects. In our scenario, the mobile device executes a DigiHome Object that offers the user preferences as context information and hosts an adaptive application enabling the control of home appliances (that also consumes events indirectly in order to be adapted). The DigiHome Core can also be considered as a DigiHome Object. Because our solution is based on standards, and in hiding the service implementation with SCA, we can easily integrate other services in the smart home that are not part of the infrastructure (in particular, the actuators). In a similar way, we are exposing the DigiHome Objects via ubiquitous bindings so that other applications (that are not part of DigiHome) can benefit from the services offered by the platform.

The exchange of events between DigiHome Objects is done following a REST-based approach. This means that we exploit the simple REST interfaces (*i.e.*, PUT, POST, DELETE and GET) and unique identifiers (*i.e.*, URLs). In particular, the objects subscribe and unsubscribe via the POST and DELETE interfaces respectively. The subscription request includes the URL that is used for sending the events. The PUT operation is used in event notification. Because of the environment dynamism, the subscriptions have a configurable expiration time. If a subscription is not renewed, it will be discarded. Thus, this simple approach enables the usage of DigiHome Objects in different kinds of devices.

12.4.3 CEP Engine

To manage the events in our scenario, we need a decision-making engine that can process them and that can create relationships to identify special situations, using predefined rules. In order to identify the desired events, the CEP Engine requires to communicate with an Event Collector, which is in charge of dealing with the subscriptions to the event sources. If an adaptation situation is detected, a corresponding action is triggered, which can go from an instruction to an actuator, to the adaptation of the system by adding or removing functionality. These actions are received by the Decision Executor, which has the responsibility of communicating with the different actuators in the environment.

Because connections problems with the event sources are possible, the CEP Engine is configured with a set of rules and actions that are applied by default. For example, if the sensor movement detects the presence of someone in the room but the DigiHome core can not detect her or his mobile phone, the system will apply a default rule for inhabitant presence and the associated actions according to the year season. These actions and

rules can be modified at deployment and runtime (thanks to the FRASCATI reconfiguration capabilities).

In DIGIHOME, for the event processing in the Controller, we use ESPER [14], a Java open source stream event processing engine, to deal with the event management and decision making process. We chose ESPER for our platform because it is the most supported open source project for CEP and is very stable, efficient, and fairly easy to use. The following code excerpt shows an example of an ESPER rule used in our scenario, in Section 12.2:

```
select sum(movement)
from MovementSensorEvent.win:time(60 sec)
```

This demonstrates the use of a time window, which is a moving interval of time. The rule collects all the events from the movement sensor from the last 60 seconds. By doing this, we can know if a user is still in the room or has already left, and adapt the room accordingly.

12.4.4 Support for Wireless Sensor Networks

In order to consume events from WSNs, we use the REMORA Component Framework [9]. This framework is an extension of SCA that brings component-based development into WSNs. The main motivation behind proposing REMORA is to facilitate *high-level and event-driven* programming in WSNs through a component-based abstraction. The primary feature of REMORA is provisioning a high-level abstraction allowing a wide range of embedded systems to exploit it at different software levels from operating systems to applications. REMORA achieves this goal by: *i)* deploying components within a lightweight framework executable on any system software written in the C language, and *ii)* reifying the concept of *event* as a first-class architectural element simplifying the development of event-oriented scenarios. REMORA meets efficiently the heterogeneity concerns related to WSN programming in the DigiHome platform as this model is SCA-compliant and portable to different operating systems used in the home sensor nodes.

REMORA proposes a TCP/IP mechanism to exchange events, which is encapsulated in an SCA component. We reuse this mechanism in order to define DIGIHOME objects for WSNs (so called Remora Objects), which are able to produce and consume simple events in the DIGIHOME platform. With these objects, we improve the efficiency of the system because the WSN is able to process simple events instead of going through the DigiHome Core for making local decisions (*e.g.* energy saving properties). In particular, WSNs are equipped with a set of rules to monitor basic events, aggregate them, and emit the inferred events as global events to the CEP engine. The local rules are essentially elicited from the global rules identified based on the requirements of home owner, avoiding any potential conflicts between global and local rules. In addition to the use of local rules for node-level decisions, the core of our framework enables in-WSN decisions, whenever an event is required to be processed with other relevant events generated by other sensor nodes. For example, if a temperature sensor detects a high temperature, it needs to become aware

of the smoke density in the room to know if there is a fire—*i.e.*, communicate with the smoke detecting sensors. Furthermore, benefiting from the DigiHome modularization of concerns, as well as the transparent communication promoted by SCA, DigiHome objects can consume/notify events from/to Remora Objects with a small effort. Finally, Remora enables DigiHome to dynamically deploy and adapt the objects running on the WSN nodes at runtime.

12.5 Empirical Validation

Although the contribution of this article lies in the adoption of a versatile architecture style for integrating the diversity of device appliances available in the pervasive environments, we have also made a performance evaluation of a prototype, implementing the proposed platform. This experimentation demonstrates the reasonable overhead imposed by the DigiHome platform.

12.5.1 Implementation Details

We built a prototype of the DigiHome platform based on FraSCAti (cf. Section 12.3.2).

The selection of this platform is motivated by two main reasons: *i)* The platform brings reflection and reconfiguration capabilities at runtime into SOA systems and, *ii)* The FraSCAti customization capabilities according to the developer needs. The former is necessary in order to enable the dynamic adaptation of DigiHome applications. The latter allows us to easily have lightweight versions of DigiHome for executing them on devices with restricted capabilities, such as the mobile devices in the smart home scenario. In order to implement the ubiquitous bindings, we have used Cyberlink for Java[1] version 1.7 for UPnP and the jSLP library[2] for SLP. More detail about the ubiquitous bindings architecture and implementation can be consulted in [10]. Once the services are discovered, the DigiHome platform uses the data bindings for interacting with them. These data bindings follow a RESTful approach in order to exchange information [15].

12.5.2 Discovery and Communication Overhead

Test Bed Configuration

In order to test DigiHome, we have employed two MacBook Pro laptops, with the following software and hardware configuration: 2.4 GHz processor, 2 GB of RAM, AirPort Extreme card, Mac OS X 10.5.6 (kernel Darwin 9.6.0), Java Virtual Machine 1.6.0, and Julia 2.5.2. The mobile client used in the tests is a Nokia N800 Internet Tablet with 400 Mhz, 128 MB of RAM, interface WLAN 802.11 b/e/g, Linux Maemo (kernel 2.6.21), CACAOVM Java Virtual Machine 0.99.4, and Julia 2.5.2.

[1] Cyberlink for Java: `http://cgupnpjava.sourceforge.net/`
[2] jSLP: `http://jslp.sourceforge.net/`

Evaluation Results

We have implemented the situation 1 of the motivating scenario (cf. Section 12.2) in order to measure the media latency for discovery (of preferences provider) and context dissemination in the DigiHome platform.

Table 12.1 reports these measures. We have executed 10000 successful tests, of which the first 100 were considered as part of the warm-up. In this setup, we retrieve the user preferences from multiple local and distributed providers and use multiple formats for the context information (*i.e.*, XML, JSON, and Java Object Serialization). In the local tests we executed the DigiHome core and the DigiHome objects in different virtual machines on the same laptop. In the distributed measures we used one laptop as Controller, and the other laptop and the Nokia device as information providers.

We also measured the delay for discovering the information provided by the sources. For discovery, we selected the UPnP and SLP protocols. In the tests, the platform aggregates the user's preferences to reduce the number of messages exchanged between the provider and the consumer. The measured time corresponds to the exchange of REST messages as well as the marshalling/unmarshalling of the information. The cost of executing others protocols, such as ACN and ZigBee was not considered in this article. The reader can find more information about the overhead introduced by these protocols in [16].

As seen, there is a linear increase of the latency with the different formats. This is a good characteristic of our solution, because we can integrate several entities with an acceptable overhead. We also observe that there is not a big variation in the communication cost between the different formats when the number of providers is low (until aprox. 10). As expected, when the providers are increased, the context exchange with object serialization is more efficient than the JSON and XML representations. Furthermore, the network usage introduces an overhead of approximately 300%.

Regarding the discovery cost (that includes the discovery time as well as the cost associated with the ubiquitous binding configuration), it is negligible compared to the context information retrieval, if there are not many providers. Our tests show that the use of SLP or UPnP for discovery does not have a big impact on the discovery time. In a similar way to the retrieval case, the measures including the network are bigger. Finally, we tested our solution using a Nokia Internet Tablet as a preference provider. As it can be seen, the use of this mobile device introduces an additional but still acceptable overhead for discovery and information exchange. This increase in cost is expected because of the limited resources of this kind of devices.

12.5.3 Event Processing Overhead

The latency for disseminating, as well as for discovering context, confirms that DigiHome can integrate heterogeneous entities with a reasonable performance overhead. Furthermore, according to the documentation provided by Esper[14], the efficiency of the engine to process the events is such that it exceeds over 500,000 events per second on a workstation

Table 12.1: Performance of the DigiHome Platform.

Number of Providers	Retrieval Latency (Local Providers)			Retrieval Latency (Remote Providers)			Discovery Latency (Local Providers)		Discovery Latency (Remote Providers)	
	Object (ms)	JSON (ms)	XML (ms)	Object (ms)	JSON (ms)	XML (ms)	SLP (ms)	UPnP (ms)	SLP (ms)	UPnP (ms)
1	**38**	31	39	**101**	108	112	8	11	34	45
2	**46**	45	62	**160**	174	179	17	22	58	75
5	**65**	90	93	**361**	377	380	39	52	119	148
10	**145**	149	309	**418**	425	444	77	102	238	315
50	**504**	556	622	**1954**	1997	2039	361	481	1060	1410
100	**933**	1449	1500	**2269**	2286	2305	730	974	2194	2912
1 (N800)	N/A	N/A	N/A	**340**	376	N/A	N/A	N/A	129	136

and between 70,000 and 200,000 events per second running on an average laptop. Thanks to the efficiency of the engine, the use of event processing in our system can be done at a low cost and given the modularity of our architecture, the Esper engine can be installed in the device that provides the highest processing power. In the context of the DigiHome platform, we observed that Esper took $1ms$ on average to process the adaptation rules.

12.6 Related Work

12.6.1 Smart Home Solutions

Thanks to the increasing popularity of smart homes in the last years, we can find several solutions dealing with the integration of services in this kind of environments. For example, in [17] the authors propose ZUMA, a middleware solution providing universality, multi-user optimality, and adaptability. The authors claim that this solution based on clean abstractions for users, content and devices makes the integration of heterogeneous entities in smart homes easier. ZUMA defines a light-weight Device Control Protocol (DCP) that all the devices in the environment have to implement in order to be used by the middleware. The discovery is done by means of a registry that is part of the platform. In DigiHome, we do not impose a single protocol for interaction or discovery. Whereas, we based our solution on standard protocols and provide the flexibility to incorporate new protocols when required. These properties, combined with the advantages from SCA, foster the building of a more versatile solution for dealing with heterogeneity in smart environments.

Gaia [18] is a distributed middleware that provides similar functionality to an operating system. Gaia defines the concept of *active spaces* as geographic regions with limited and well-defined physical boundaries containing physical objects, heterogeneous networked devices, and users performing a range of activities. Examples of active spaces include meeting rooms and smart homes. The platform allows the coordination of software entities and

heterogeneous devices in these active spaces. To do it, GAIA provides services for event management and distribution, context information query (for the context-based adaptation of applications), detection of digital and physical entities, storage of the information associate with entities, and file management. However, despite GAIA richness in terms of services, the solution is complex and the incorporation of new entities remains difficult. The simplicity of DIGIHOME makes this task easy. Furthermore, GAIA lacks support for restricted devices, such as the mobile devices in our scenario, in contrast to DIGIHOME that uses SCA to do it.

In [19], the MAVHOME (Managing an Intelligent Versatile Home) project is described. This project aims to build a home as a rational agent that maximizes inhabitants comfort and reduces operation costs. The project mainly focuses on prediction algorithms to guide decisions for controlling devices throughout the home. These algorithms include the *Smarthome Inhabitants Prediction* algorithm (that matches more recent sequences of events with stored sequences), *Active LeZi* algorithm (that applies information theory principles to process historical actions sequences) and a Task-Based Markov Model algorithm (for identifying high level tasks in action sequences). Benefiting from the DIGIHOME extensibility and concern isolation, these algorithms can be incorporated into the CEP engine in order to make our service-oriented platform more intelligent and autonomous in respect to the adaptation decisions.

12.6.2 Context Dissemination

In literature, it is possible to find two kinds of solutions that deal with context integration: *centralized* and *decentralized*. In the centralized category we can find middleware, such as PACE [20] and *Context Distribution and Reasoning* (ConDoR) [21]. PACE proposes a centralized context management system based on repositories. The context-aware clients can interact with the repositories using protocols, such as Java RMI or HTTP. For its part, ConDoR takes advantage of the object-oriented model and ontology-based models to deal with context distribution and context reasoning, respectively. In ConDoR, the integration of the information using different protocols is not considered as an important issue. The problem with this kind of approach is the introduction of a single point of failure into the architecture, which limits its applicability to ubiquitous computing environments.

On the other hand, in decentralized approaches we can find solutions like CORTEX [22] and MUSIC [23, 24]. CORTEX defines sentient objects as autonomous entities that have the capacity of retrieving, processing, and sharing context information using HTTP and SOAP. MUSIC middleware is another decentralized solution that proposes a peer-to-peer infrastructure dealing with context mediation. The decentralized approaches face the problem of fault tolerance by distributing the information across several machines. However, as well as some centralized solutions, the lack of flexibility in terms of the communication protocols remains a key limitation for these approaches. In addition to that, peer-to-peer approaches have performance and security problems. In DIGIHOME, we provide a

solution, where the different interacting devices can process the events retrieved from the environment. Furthermore, in DigiHome we provide flexibility in terms of interaction by supporting different kinds of communication protocols and we also allow spontaneous interoperability.

12.6.3 Complex Event Processing

Given the increasing interest to integrate the flow of data into the existing systems, CEP has gained some attention as it can help to provide that integration transforming isolated data into valuable information. In this context we can find some works similar to ours in [25] and [26]. In [25], the authors integrate CEP into their existing project called SAPHE (*Smart and Aware Pervasive Healthcare*), and also use Esper as their CEP engine. As the project name shows, the project is applied to healthcare and uses sensors to monitor a patient's activity and vital signs. They use CEP to correlate and analyze the sensor data in order to calculate critical factors of the patient locally in their set-top box, without having to send all the events to an external server. In their approach they lack a way to discover new services and they never mention how, if possible, would they interact with actuators in order to adapt to the context and respond to a specific situation.

An *Event-Driven Architecture* (EDA) that combines the advantages of WSN with CEP is presented in [26]. They use an extension of the RFID EPCglobal architecture which allows the interaction of RFID and WSN events. Once the events are collected, they use CEP to detect specific situations. They use a smart shelf application as their scenario to show how the events from both sources can be combined. Even though both technologies seem to interact in their project, their specification is somehow limited because they do not specify how the information obtained could be used, other than generating a report that will be logged in the EPCIS server.

12.6.4 Wireless Sensor Networks

In [27], the authors describe a WSN-specialized resource discovery protocol, called DRD. In this approach, each node sends a binary XML description to another node that has the role of *Cluster Head* (CH). The CH is selected among all the nodes based on their remaining energy. Therefore, it is necessary to give all the nodes the capacity of being a CH. Consequently, all the nodes need an SQLlite database, libxml2 and a binary XML parser in order to implement the CH functionalities. In DigiHome, with our modular architecture, we consider the resource constraint of sensors nodes and provide a lightweight version of the platform based on the REMORA Framework that delegates complex processing to more powerful devices. Therefore, not all the nodes have to be CH. Furthermore, we benefit from the advertisement capacities of the sensor nodes to identify adaptation situations.

In CoBIs [28], business applications are able to access functionalities provided by the sensor nodes via web services. The major aim of the CoBIs middleware is to mediate

service requests between the application layer and the device layer. The focus lies thereby on deployment and discovery of required services.

AGIMONE [29] is a middleware solution supporting the integration of WSNs and IP networks. It focuses on the distribution and coordination of WSN applications across WSN boundaries. AGIMONE integrates the AGILLA [30] and LIMONE [31] middleware platforms. AGIMONE is a general-purpose middleware with a uniform programming model for applications, that integrates multiple WSNs and the IP network. In our approach, we also promote the integration of sensor nodes via SCA bindings. Moreover, we enable spontaneous communications with some sensor nodes that execute a lightweight version of DIGIHOME.

12.7 Conclusions and Future Work

In this article, we have presented DIGIHOME, a platform addressing the mobility, heterogeneity, and adaptation of smart entities. In particular, DIGIHOME detects adaptation situations by integrating context information using an SCA-based architecture. This architecture promotes the modularization of concerns and fosters the application of the REST principles by exploiting the SCA extensibility. The simplicity and data orientation of REST, combined with the SCA independence of implementation technologies, make DIGIHOME an attractive solution to deal with heterogeneity in terms of interactions. The definition and application of ubiquitous bindings in the platform enable spontaneous communication by means of standard protocols (e.g.,UPnP and SLP), and furnish context provider selection (based on QoC attributes). On the other hand, the modularized architecture of DIGIHOME allows the definition of variants for the platform, called DIGIHOME objects, that can be deployed on resource-constrained devices. The functionality of these objects is exposed as services, accessible via several protocols, which can be accessed by clients that do not have to be part of the platform. Furthermore, the clear separation of concerns in the DIGIHOME architecture encourages the exploitation of WSNs for simple processing and local decision making. The suitability of our platform for context integration was evaluated with different discovery and context representations.

Future work includes further tests using some sensor nodes as context information providers, which will execute Remora objects objects. Also, we are currently working on a distributed CEP approach to diminish risk of failure, given the single instance of event processing in our project. We will also define more complex scenarios including user preferences conflicts and the absence of mobile devices when the presence of someone is detected in home. These scenarios will allow a further illustration of the potential of the platform. Furthermore, we plain to incorporate a replication mechanism of the DigiHome core by benefiting from the different devices such as desktops, laptops and set-top-box available in home. This mechanism will enable the platform to deal with possible failures of the device hosting the core. Finally, we plan to exploit the FRASCATI's reconfiguration capabilities in order to integrate new communication and discovery protocols at runtime.

Acknowledgement. This work is partly funded by the EGIDE Aurora and INRIA SeaS research initiatives.

References

[1] D. Romero, G. Hermosillo, A. Taherkordi, R. Nzekwa, R. Rouvoy, and F. Eliassen, "Restful integration of heterogeneous devices in pervasive environments," in *DAIS '10: Proc. of the 10th IFIP Int. Conf. on Distributed Applications and Interoperable Systems*, ser. LNCS, vol. 6115. Springer, 2010, pp. 1–14.

[2] Open SOA, "Service Component Architecture Specifications," Nov. 2007.

[3] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, Irvine, 2000.

[4] D. C. Luckham, *The Power of Events: An Introduction to Complex Event Processing in Distributed Enterprise Systems.* Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001.

[5] L. Seinturier, P. Merle, D. Fournier, N. Dolet, V. Schiavoni, and J.-B. Stefani, "Reconfigurable sca applications with the frascati platform," in *SCC '09: Proc. of the IEEE Int. Conf. on Services Computing.* Washington, DC, USA: IEEE Computer Society, Sep. 2009, pp. 268–275.

[6] R. Mélisson, P. Merle, D. Romero, R. Rouvoy, and L. Seinturier, "Reconfigurable runtime support for distributed service component architectures," in *Automated Software Engineering, Tool Demonstration*, Belgium, 2010.

[7] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The Fractal component model and its support in java," *Soft. Pract. and Exp.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[8] *Listeners Considered Harmful: The Whiteboard Pattern*, OSGi Alliance, 2004.

[9] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. L. Trung, and F. Eliassen, "Programming sensor networks using Remora component model," in *DCOSS '10: Proc. of the 6th Int. Conf. on Distributed Computing in Sensor Systems.* Santa Barbara, CA, USA: Springer, 2010, pp. 45–62.

[10] D. Romero, R. Rouvoy, L. Seinturier, and P. Carton, "Service discovery in ubiquitous feedback control loops," in *DAIS '10: Proc. of the 10th IFIP Int. Conf. on Distributed Applications and Interoperable Systems*, ser. LNCS, vol. 6115. Springer, june 2010, pp. 113–126.

[11] UPnP Forum, "UPnP Device Architecture 1.0," http://www.upnp.org/resources/documents.asp, Apr. 2008.

[12] E. Guttman, C. Perkins, J. Veizades, and M. Day, "Service Location Protocol, Version 2. RFC 2608 (Proposed Standard)," http://tools.ietf.org/html/rfc2608, Jun. 1999.

[13] M. Krause and I. Hochstatter, "Challenges in modelling and using quality of context (qoc)," in *Proc. of the 2nd Int. Workshop on Mobility Aware Technologies and Applications*, Montreal, Canada, 2005, pp. 324–333.

[14] EsperTech, "Esper," http://esper.codehaus.org.

[15] D. Romero, R. Rouvoy, L. Seinturier, and F. Loiret, "Integration of heterogeneous context resources in ubiquitous environments," in *SEAA '10: Proc. of the 36th EUROMICRO Int. Conf. on Software Engineering and Advanced Applications*. Lille, France: ACM, 2010, p. 4.

[16] Zigbee Alliance, "ZigBee and Wireless Radio Frequency Coexistence," http://www.zigbee.org/imwp/download.asp?ContentID=11745, Jun. 2007.

[17] C. Baker, Y. Markovsky, J. Greunen, J. Rabaey, J. Wawrzynek, and A. Wolisz, "Zuma: A platform for smart-home environmnents," in *Intelligent Environments, 2006. IE 06. 2nd IET Int. Conf. on*, vol. 1, july 2006, pp. 51 –60.

[18] M. Román, C. Hess, R. Cerqueira, A. Ranganathan, R. H. Campbell, and K. Nahrstedt, "A middleware infrastructure for active spaces," *IEEE Pervasive Computing*, vol. 1, no. 4, pp. 74–83, 2002.

[19] D. J. Cook, M. Youngblood, E. O. Heierman, K. Gopalratnam, S. Rao, A. Litvin, and F. Khawaja, "Mavhome: An agent-based smart home," in *PERCOM '03: Proc. of the 1st IEEE Int. Conf. on Pervasive Computing and Communications*. Washington, DC, USA: IEEE Computer Society, 2003, p. 521.

[20] K. Henricksen, J. Indulska, and T. Mcfadden, "Middleware for distributed context-aware systems," in *DOA '05: Int. Symposium on Distributed Objects and Applications*. Springer, Nov. 2005, pp. 846–863.

[21] F. Paganelli, G. Bianchi, and D. Giuli, "A context model for context-aware system design towards the ambient intelligence vision: Experiences in the etourism domain," in *Universal Access in Ambient Intelligence Environments*, 2006, pp. 173–191.

[22] C.-F. Sorensen, M. Wu, T. Sivaharan, G. S. Blair, P. Okanda, A. Friday, and H. Duran-Limon, "A context-aware middleware for applications in mobile ad hoc environments," in *MPAC '04: Proc. of the 2nd Workshop on Middleware for Pervasive and Ad-hoc Computing*. Toronto, Canada: ACM, 2004, pp. 107–110.

[23] X. Hu, Y. Ding, N. Paspallis, P. Bratskas, G. A. Papadopoulos, P. Barone, and A. Mamelli, "A peer-to-peer based infrastructure for context distribution in mobile and ubiquitous environments," in *CAMS '07: Proc. of 3rd Int. Workshop on Context-Aware Mobile Systems*, Algarve, Portugal, 2007.

[24] M. Kirsch-Pinheiro *et al.*, "Context grouping mechanism for context distribution in ubiquitous environments," in *DOA '08: Proc. of the OTM Int. Conf. on Distributed Objects and Applications*. Monterrey, Mexico: Springer, 2008, pp. 571–588.

[25] G. E. Churcher and J. Foley, "Applying and extending sensor web enablement to a telecare sensor network architecture," in *COMSWARE '09: Proc. of the 4th Int. ICST Conf. on COMmunication System softWAre and middlewaRE*. New York, NY, USA: ACM, 2009, pp. 1–6.

[26] W. Wang, J. Sung, and D. Kim, "Complex event processing in epc sensor network middleware for both rfid and wsn," in *ISORC '08: Proc. of the 11th IEEE Symposium on Object Oriented Real-Time Distributed Computing*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 165–169.

[27] S. Tilak, N. A.-G. K. Chiu, and T. Fountain, "Dynamic resource discovery for wireless sensor networks," 2005.

[28] COBIS Consortium, "Cobis. fp strep project ist 004270," 2009, http://www.cobis-online.de.

[29] G. Hackmann, C.-L. Fok, G.-C. Roman, and C. Lu., "Agimone: Middleware support for seamless integration of sensor and ip networks," in *DCOSS '06: Int. Conf. on Distributed Computing in Sensor Systems*. Springer, 2006.

[30] C.-L. Fok, G.-C. Roman, and C. Lu, "Agilla: A mobile agent middleware for self-adaptive wireless sensor networks," *ACM Trans. Auton. Adapt. Syst.*, vol. 4, 2009.

[31] C.-L. Fok, G.-C. Roman, and G. Hackmann, "A lightweight coordination middleware for mobile computing," in *Coordination '04: Proc. of the 6th Int. Conf. on Coordination Models and Languages*. Springer, 2006, pp. 135–151.