KATHOLIEKE UNIVERSITEIT
LEUVEN

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science

Management solutions for distributed software applications in multi-purpose sensor networks

Wouter Horré

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2011

Management solutions for distributed software applications in multi-purpose sensor networks

Wouter Horré

Jury:
Prof. dr. ir. E. Aernoudt, president
Prof. dr. ir. W. Joosen, promotor
Prof. dr. ir. P. Verbaeten
Prof. dr. ir. R. Lauwereins
Dr. S. Michiels
Prof. dr. D. Hughes
Prof. dr. G. Coulson
(Lancaster University, United Kingdom)

Dissertation presented in partial fulfillment of the requirements for the degree of Doctor in Engineering

October 2011

D/2011/7515/125
ISBN 978-94-6018-424-6

Preface

This dissertation describes the results of five years of work. Results which would not have been possible without the direct and indirect support of several people and organizations. I would like to sincerely thank these people and organizations for their financial, scientific, administrative, technical and/or personal support.

The majority of this work was supported by a PhD fellowship of the Research Foundation - Flanders (FWO). I would like to express my gratitude to the FWO for their financial support.

Thanks to the members of my jury for the insightful and interesting discussion at the preliminary defense and for the helpful comments on this dissertation. I thank Wouter Joosen for being my promotor and guiding me through the PhD trajectory. Sam Michiels, I am grateful for your guidance and the numerous inspiring and motivating discussions. Thank you, Danny Hughes, for your enthusiasm I experienced during our collaborations, both local and remote. I thank Pierre Verbaeten and Rudy Lauwereins for being the assessors of my PhD trajectory. Special thanks to Geoff Coulson for accepting the invitation to be an external member of the jury and to Etienne Aernoudt for chairing the jury.

Research is often a team effort, therefore I thank all my current and former colleagues at the Department of Computer Science and DistriNet in particular. Special thanks to the members of the networking task force for the countless collaborations, discussions and informal chats. I also thank all my former and current office mates and lunch mates for the smalltalk on various subjects. I am also thankful to the administrative and technical staff of DistriNet and the department for their support.

This dissertation would have never existed without the support of my friends and family that helped me relax and recharge in the evenings, weekends and holidays. Thank you for that. A very special thanks to my parents and sisters for their unconditional support. And last but not least: Saartje, thank you for everything!

Wouter Horré, September 2011

Abstract

Multi-purpose sensor networks represent an emerging use case of wireless sensor networks. Multi-purpose sensor networks no longer serve a single purpose, but are a reusable asset that host multiple applications for a variety of actors. On the one hand, multi-purpose sensor networks have the potential to increase the return on investment of wireless sensor networks. On the other hand, multi-purpose sensor networks introduce additional management complexity that may increase the operational expenses. Automated management solutions are an important tool to reduce these operational expenses.

Management solutions for multi-purpose sensor networks must meet four requirements: (1) resource efficiency, (2) autonomy, (3) reconfigurability and (4) multi-application support. The first requirement stems from the resource constraints of wireless sensor nodes. Management solutions must take these constraints into account. Second, to avoid the need for manual intervention due to the inherent dynamism of sensor networks, the management solutions must autonomously handle network changes. Third, to handle these changes and accommodate the evolving end-user requirements, support for flexible reconfiguration is indispensable. Finally, multi-application support is essential to facilitate multi-purpose usage of the sensor network.

To address resource efficiency, the research in this dissertation leverages the resources in the gateway and back-end tiers that interact with the multi-purpose sensor network. The presented approach facilitates autonomy and eases multi-application support by raising the abstraction level of the policies that govern the management solutions. Modularization of the software on the sensor nodes is applied to improve reconfigurability and to enable multi-application support through distributed application assembly.

This dissertation presents two contributions to the state-of-the-art in runtime support for reconfigurability. Both contributions provide flexible reconfiguration and a foundation for multi-application support through modularity. The first contribution, an adaptable virtual machine (DAViM), focuses on maximizing the modularity and reconfigurability of the runtime itself. The second contribution,

a component-based approach (LooCI), provides modularity both in the runtime and at application development time. DAViM and LooCI focus on different points in the trade-off between fine-grained reconfiguration (DAViM) and clean modularization (LooCI).

Next, this dissertation describes a management solution (QARI) for assembly and deployment of distributed software applications that provides autonomy and full multi-application support on top of a modular runtime system. QARI supports a high-level abstraction that allows administrators to easily express the goals for distributed application deployment. QARI autonomously pursues these goals and manages the trade-off between multiple applications. QARI integrates with LooCI because the cleaner modularization of LooCI's component-based approach eases application assembly and promotes component reuse.

Prototype implementations of DAViM, LooCI and QARI confirm the feasibility of the approach for resource constrained wireless sensor networks. Two representative application scenarios in transport and logistics illustrate the features of the solutions. Finally, the text reports on the integration of the prototypes of QARI and LooCI in a testbed with state-of-the-art sensor node hardware.

Samenvatting

Draadloze sensornetwerken worden steeds vaker ingezet voor meerdere doeleinden. Bij dergelijk gebruik dient het sensornetwerk niet langer één enkel doel, maar wordt het beschouwd als herbruikbare infrastructuur die meerdere toepassingen host voor verschillende actoren. Aan de ene kant hebben sensornetwerken voor meerdere doeleinden het potentieel om het financieel rendement (de *return on investment*) van sensornetwerken te verhogen. Aan de andere kant introduceert het gebruik voor meerdere doeleinden extra beheerscomplexiteit die de operationele kosten kan doen stijgen. Geautomatiseerde beheersoplossingen zijn een belangrijk middel om deze operationele kosten te verlagen.

Beheersoplossingen in sensornetwerken voor meerdere doeleinden moeten voldoen aan vier vereisten: (1) efficiënt gebruik van hulpbronnen, (2) autonomie, (3) herconfigureerbaarheid en (4) ondersteuning voor meerdere toepassingen. De eerste vereiste is een gevolg van de beperkte hulpbronnen die sensorknopen ter beschikking hebben. Beheersoplossingen moeten met deze beperkingen rekening houden. Ten tweede moeten de beheersoplossingen veranderingen in het netwerk autonoom afhandelen om de nood voor manuele interventies die voortvloeien uit de inherente dynamiek in sensornetwerken te vermijden. Ten derde is ondersteuning voor flexibele herconfiguratie onontbeerlijk om deze veranderingen te kunnen afhandelen en om te kunnen reageren op de evoluerende vereisten van eindgebruikers. Tenslotte is het essentieel dat er ondersteuning voor meerdere toepassingen wordt voorzien om het sensornetwerk ten volle te kunnen inzetten voor meerdere doeleinden.

Om efficiënt gebruik van de hulpbronnen te bewerkstelligen, steunt het onderzoek in dit doctoraat op de hulpbronnen in de gateway en back end infrastructuur die interageert met het sensor netwerk. De voorgestelde aanpak maakt autonomie mogelijk en vergemakkelijkt ondersteuning voor meerdere toepassingen door het abstractieniveau te verhogen van de policies die het gedrag van de beheersoplossingen sturen. Modularisatie van de software op de sensorknopen verbetert de herconfigureerbaarheid en maakt ondersteuning voor meerdere toepassingen mogelijk door middel van gedistribueerde compositie van toepassingen.

Deze doctoraatsthesis beschrijft twee contributies tot de state-of-the-art in ondersteuning voor herconfiguratie in de uitvoeringsomgeving. Beide contributies voorzien flexibele herconfiguratie en basisondersteuning voor meerdere toepassingen door middel van modulariteit. De eerste contributie, een aanpasbare virtuele machine (DAViM) spitst zich toe op het maximaliseren van de modulariteit en herconfigureerbaarheid van de uitvoeringsomgeving zelf. De tweede contributie, een componentgebaseerde aanpak (LooCI), biedt modulariteit zowel in de uitvoeringsomgeving als tijdens de ontwikkeling van een toepassing. DAViM en LooCI richten zich op twee verschillende punten in de afweging tussen fijnkorrelige herconfiguratie (DAViM) en propere modulariteit (LooCI).

Verder beschrijft deze thesis een beheersoplossing (QARI) voor de compositie en installatie van gedistribueerde softwaretoepassingen die autonomie en volledige ondersteuning voor meerdere toepassingen aanbiedt steunend op een modulaire uitvoeringsomgeving. QARI ondersteunt een hoog abstractieniveau dat beheerders toelaat om gemakkelijk de doelen voor de installatie van gedistribueerde toepassingen te beschrijven. QARI streeft deze doelen autonoom na en beheert de afweging tussen meerdere toepassingen. QARI integreert met LooCI omdat de propere modulariteit van LooCI's componentgebaseerde aanpak compositie van toepassingen makkelijker maakt en hergebruik van componenten aanmoedigt.

Prototype implementaties van DAViM, LooCI en QARI bevestigen de haalbaarheid van de voorgestelde aanpak in draadloze sensornetwerken met beperkte hulpbronnen. Twee representatieve toepassingsscenario's in transport en logistiek illustreren de mogelijkheden van de oplossingen. Tenslotte gaat de tekst dieper in op de integratie van de prototypes van QARI en LooCI in een testopstelling met state-of-the-art sensorknopen.

Contents

Contents vii

List of Figures xiii

List of Tables xv

List of Listings xvii

1 Introduction and problem statement 1

1.1 Wireless sensor networks and their applications 1

1.1.1 History of wireless sensor networks 2

1.1.2 Characteristics of wireless sensor nodes and networks 3

1.1.3 Applications of wireless sensor networks 4

1.2 Multi-purpose sensor networks 5

1.3 Problem statement 6

1.4 Contributions 8

2 Requirements, approach and overview 11

2.1 Requirements 12

2.2 Related work 13

2.2.1 Autonomic computing 13

2.2.2 Dynamic change management 15

2.2.3 Multi-application support 17
2.3 Approach 18
2.3.1 Leverage resources in gateway and back-end tiers 19
2.3.2 Raising the abstraction level 20
2.3.3 Modularization 21
2.4 Key contributions 22
2.5 Summary 26

3 Runtime support for reconfiguration 27
3.1 State-of-the-art in dynamic reconfiguration of wireless sensor networks 28
3.2 An adaptable virtual machine for sensor networks 31
3.3 Supporting modularity at development time 34
3.4 A component model for multi-purpose sensor networks 38
3.4.1 The component model 38
3.4.2 Reconfiguration 40
3.4.3 Distribution 43
3.4.4 Loose coupling 44
3.5 Discussion 44
3.6 Summary 46

4 Application management 49
4.1 Related work 50
4.1.1 Abstraction level and degree of automation 51
4.1.2 Utility in wireless sensor networks 57
4.2 Raising the abstraction level 58
4.2.1 Stakeholders 59
4.2.2 Network information 61
4.2.3 Application information 62

4.2.4 Deployment information 63
4.3 Multi-application support 65
4.4 Architecture 66
4.4.1 Context 66
4.4.2 Architecture overview 67
4.4.3 Architecture details 69
4.5 Discussion 74
4.5.1 Planning deployment 75
4.5.2 Executing deployment 78
4.6 Summary 79

5 Prototype, illustration and evaluation 81

5.1 Example scenarios 82
5.1.1 Application context: transport and logistics 82
5.1.2 Example application scenarios 83
5.1.3 Illustration of the policy specifications 86
5.2 DAViM implementation 89
5.2.1 Illustration 89
5.2.2 Implementation and evaluation 90
5.3 LooCI implementation 93
5.3.1 Rationale 93
5.3.2 Developing a LooCI component 95
5.3.3 Reconfiguration of LooCI nodes 97
5.3.4 Implementation and evaluation 98
5.4 QARI implementation 101
5.4.1 Rationale 102
5.4.2 Illustration 102
5.4.3 Implementation and evaluation 108

5.5 Integration . . . 111
5.6 Summary . . . 115

6 Conclusion and future work **117**
6.1 Contributions . . . 117
6.2 Future work . . . 119
6.3 Future outlook . . . 120

A Policy specifications: XML schemas **123**
A.1 Composition specification: LooCI SCA extensions . . . 123
A.2 Network specification . . . 124
A.3 Constraint specification . . . 125
A.4 Deployment specification . . . 126

B LooCI/Contiki API **129**
B.1 The peer library . . . 129
B.1.1 Define Documentation . . . 129
B.1.2 Function Documentation . . . 130
B.2 Event types . . . 131
B.2.1 Typedef Documentation . . . 131
B.2.2 Function Documentation . . . 132
B.3 Components . . . 132
B.3.1 Data Structure Documentation . . . 132
B.3.2 Enumeration Type Documentation . . . 132
B.3.3 Define Documentation . . . 133
B.4 Events . . . 135
B.4.1 Data Structure Documentation . . . 135
B.4.2 Define Documentation . . . 135
B.4.3 Function Documentation . . . 136

C LooCI/Contiki reconfiguration and introspection API 139

C.1 Common definitions . . . 139

C.1.1 Define Documentation . . . 139

C.2 Deployment . . . 139

C.2.1 Function Documentation . . . 139

C.3 Runtime control . . . 140

C.3.1 Function Documentation . . . 140

C.4 Introspection . . . 146

C.4.1 Data Structure Documentation . . . 146

C.4.2 Define Documentation . . . 146

C.4.3 Enumeration Type Documentation . . . 146

C.4.4 Function Documentation . . . 147

C.5 Shell wrapper . . . 151

Bibliography 153

List of scientific publications 169

List of Figures

2.1 The MAPE-K reference architecture for autonomic computing [71] 14

2.2 Decomposition view of the overall approach. 24

2.3 Deployment view of the overall approach. 25

3.1 Positioning of the work on runtime support for reconfiguration. . . 28

3.2 High-level overview of the DAViM architecture. 33

3.3 Illustration of the important concepts of LooCI, a component model for multi-purpose sensor networks. 39

3.4 The life-cycle of a LooCI component. 40

4.1 Positioning of the application management solution (QARI). . . . 50

4.2 Overview of the level of abstraction and the degree of automation in state-of-the-art management tools for distributed applications. . 53

4.3 The domain model for the composition specification, constraint specification, deployment specification and network description. . . 60

4.4 Model of the infrastructural context in which QARI operates. . . . 66

4.5 System boundaries for QARI. 67

4.6 Overview of QARI 68

4.7 Detailed view of QARI's architecture. 70

4.8 Pseudocode for a heuristic algorithm to solve the simplified planning problem. 76

5.1 Positioning of the prototypes of DAViM, LooCI and QARI. 82

5.2 Illustration of a sensor network in a truck/trailer combination. . . 85

5.3 Overview of the LooCI runtime implementation on Contiki. 99

5.4 Area covered by node A with a sensing range of 2. 102

5.5 Calculated deployment patterns for a single component type using the algorithm in Figure 4.8. 103

5.6 Deployment patterns for a shared network. 104

5.7 Minimal and average coverage over time for the temperature sampling component in the presence of random node failures. . . . 104

5.8 Example result of QARI's planning for the components on the trailer in the truck/trailer monitoring scenario. 105

5.9 QARI Management console: node view. 107

5.10 QARI Management console: goal view. 108

5.11 QARI Management console: coverage view. 109

5.12 Coverage view in the QARI management console for a temperature sampling component on the testbed. 113

5.13 Goal view in the QARI management console for the multi-application experiment on the testbed. 114

List of Tables

3.1 Comparison of component-based approaches for wireless sensor networks 37

5.1 Evaluation of DAViM 92

5.2 Flash and RAM size in bytes of a temperature sampling application in Contiki and LooCI. 100

5.3 Planning time for the truck/trailer monitoring scenario. 111

List of Listings

3.1 The deployment API of LooCI. 41

3.2 The runtime control API of LooCI. 41

3.3 The introspection API of LooCI. 42

4.1 Pseudocode for interface: IChangeNotification. 71

4.2 Pseudocode for interface: IApplicationManagement. 72

4.3 Pseudocode for interface: IPlanningStrategy. 72

4.4 Pseudocode for interface: IExecutionPlugin. 73

4.5 Pseudocode for interface: INetworkInformation. 74

5.1 Network description for the truck/trailer scenario. 86

5.2 Constraint specification for the truck/trailer scenario. 87

5.3 Composition specification for a temperature monitoring application in the truck/trailer scenario. 87

5.4 Deployment specification for a temperature monitoring application and a humidity monitoring application in the truck/trailer scenario. 88

5.5 Example LooCI component for temperature filtering. 94

5.6 Example of using the shell wrapper around the reconfiguration API to manage a LooCI node. 97

5.7 Network specification for the testbed of AVR Raven sensor nodes. 112

Chapter 1

Introduction and problem statement

This dissertation concerns management solutions for distributed software applications in multi-purpose sensor networks. This introduction first discusses wireless sensor networks and their applications in Section 1.1. Section 1.2 continues with a description of multi-purpose sensor networks. Next, Section 1.3 presents the problem statement of this dissertation. This chapter concludes with an overview of the key contributions and an outline for the rest of the text in Section 1.4.

1.1 Wireless sensor networks and their applications

Wireless sensor networks are networks of cheap, small, low-power devices with multiple sensors, connected through wireless links. Wireless sensor nodes provide a combination of sensing, processing and communication leading to several opportunities for accurate instrumentation of the physical world. Potential applications span a broad range of domains: environmental and habitat monitoring, industrial sensing, home automation, building security, traffic control and more.

Instrumentation of the physical world has long been within human interest for several reasons. First, the numbers tell the tale: in order to better understand physical phenomena, collecting accurate data about the phenomena is crucial. Second, people strive towards an intelligent environment, visions such as the *intelligent fridge* [121], the *intelligent room* [18] and ubiquitous computing in

general [142] result from this desire. Accurate information about the environment is crucial for such intelligent behavior.

Wireless sensor networks allow applications to gather information about their physical environment. This enables applications to integrate with the physical environment in which they operate. Wireless sensor networks thus provide mechanisms to bridge between the physical world and the virtual world. Wireless sensor networks can therefore also be used in Cyber-Physical Systems (CPS) [89], systems integrating computation and physical processes.

First, Section 1.1.1 presents a historical perspective on wireless sensor networks. Section 1.1.2 continues with a discussion on the typical characteristics of modern sensor nodes and sensor networks. Finally, Section 1.1.3 introduces some of the potential applications.

1.1.1 History of wireless sensor networks

Even before the term *sensor network* was coined, prototypical sensor network systems existed [23]. Initially, military applications have been a driver for the research and development of such systems. For example, during the cold war, a system of acoustic sensors on the ocean bottom (SOSUS) was deployed for tracking submarines. The cold war also stimulated development and deployment of networks of air defense radars. The research during this early phase focused on the needs of the applications, e.g. acoustic signal processing, but nevertheless provided early results for the development of modern sensor networks.

The Distributed Sensor Networks (DSN) program at the Defence Advanced Research Projects Agency (DARPA) in the United States of America started modern research on sensor networks around 1980 [23]. The components identified for a DSN were sensors, communication, processing techniques and algorithms, and distributed software. Two demonstrators were developed in this program: a testbed for acoustic tracking of a low-flying aircraft [87] and a distributed vehicle monitoring testbed [25].

Although the technology for cheap, small sensor nodes was not yet available, several military sensor networks based on the results of the DSN program were developed in the 1980s and 1990s [23]. The combination of advances in micro-electromechanical systems (MEMS), wireless networking and low-power processors in the late 1990s did enable these small sensor nodes and led to a significant shift in sensor network research. These technologies allowed to align the research closer with the original vision of large-scale networks of small devices. For example, the Smart Dust research project [79] explored the limits of size and power consumption in sensor nodes.

The availability of low-cost sensor nodes and communication networks fueled the research on sensor networks for a vast range of other applications besides the original military applications (See Section 1.1.3). This resulted in significant advances in technology, runtime systems, communication protocols, programming abstractions and services for wireless sensor networks [147, 139].

As the field matures, standardization efforts emerged. First, standards for the physical and MAC layer were developed [31, 72]. Later on, the standardization moved up in the network stack, with standardization efforts in the network layer [67], in routing [74], and in the application layer [73].

1.1.2 Characteristics of wireless sensor nodes and networks

The low-cost sensor nodes that are available for research and commercial applications combine three basic capabilities: sensing, processing and (wireless) communication. Sensor nodes are typically battery operated and as a consequence the processing and communication resources are limited to remain low-power. Sensor nodes with varying resources are available. Low-end sensor nodes—often called motes or mote-class sensor nodes—are designed to last years on a single battery charge. They have capabilities as low as an 8-bit MCU, a few kB of RAM and tens of kB of program memory. Examples of such platforms are AVR Raven [7] and TelosB [103]. High-end sensor nodes on the other hand often have access to harvested energy or they are accessible for recharging. These nodes are built using 32-bit CPUs and up to MBs of RAM and program memory. Examples of this type of devices are iMote2 [30] and SunSPOT [112].

Communication drains a large amount of energy from sensor nodes' battery [42]. Therefore sensor nodes are equipped with radios that implement low-power protocols that have been specifically designed (or adapted) for wireless sensor networks. These tend to have low throughputs and relatively short communication ranges. For example, IEEE 802.15.4 [72], one of the often used communication standards in wireless sensor networks provides a maximum speed of 250 kbit/s and the communication range is typically in the order of 10 m to 100 m. Integration with existing networks requires communication gateways that bridge between the different networking technologies.

Because of the battery powered operation, wireless communication and deployment into harsh environments, node and communication failure are common. Nodes might fail at any time due to battery depletion or environmental factors. Communication might fail—intermittently or permanently—due to for example large metal objects blocking the wireless signals. In combination with the potential mobility of sensor nodes and the tight coupling with the physical world, this results in a very dynamic environment with frequent network topology changes.

The limited communication range of sensor nodes forces the use of multi-hop networking techniques. Depending on the routing algorithm, multiple network topologies are possible. The most common protocols are variants of tree-based routing, ad-hoc routing, geographic routing and broadcast or epidemic data dissemination [92]. Hybrid approaches are also in use, for example a fully-meshed network of high-end nodes that each serve as a sink for a tree-based network of low-end nodes [53]. In any case, the protocols must take the dynamic environment and the network topology changes into account.

To overcome the limited reliability of a single sensor node, sensor networks rely on redundancy to provide reliable and accurate sensing [122]. As a consequence, sensor networks tend to be large-scale to provide sufficient redundancy. Redundancy also enables load-balancing of monitoring and computation tasks to conserve energy.

To conclude, wireless sensor networks typically exhibit the following characteristics:

- The available resources are limited. This includes processing, memory, communication and energy resources.
- The nodes and communication links have limited reliability.
- A wireless sensor network is a very dynamic environment due to the limited reliability, the potential node mobility and the tight coupling with the physical environment.
- Sensor networks tend to be large-scale and redundant to provide sufficient accuracy and reliability.

1.1.3 Applications of wireless sensor networks

The emergence of low-cost wireless sensor nodes shifted the applications of sensor network technology from military only towards a broad range of applications (Section 1.1.1). These new applications address societal, scientific, industrial or consumer needs. These applications fit into the vision of the Internet-of-Things [22], a broader trend to enhance ubiquitous objects with networking capabilities. The following paragraphs shed their light on some of the potential applications.

Environmental monitoring, e.g. flood monitoring [55, 65] or volcano [143] monitoring, addresses the societal need for early warning systems. Such systems facilitate timely evacuation that saves human lives. Sensor networks for habitat monitoring, such as the deployment on Great Duck Island for monitoring seabird colonies [143], provide essential data to derive new scientific insights into the

habitat of animals. In health care, body area sensor networks [135] have been used to remotely monitor, for example, heart rate and blood pressure.

The emergence of the ZigBee standard, a communication stack on top of IEEE 802.15.4, has driven the adoption of wireless sensor networks for home and building automation [144]. These sensor networks are used for a variety of applications situated in the traditional home automation areas such as comfort, convenience and safety, but also in areas such as energy awareness and saving. The ZigBee standard has also been popular in industrial monitoring and control applications.

This dissertation draws inspiration from amongst others the transport and logistics domain. In that context, sensor networks could be deployed throughout the supply chain in order to provide end-to-end visibility. These sensor networks facilitate development of applications for cold chain monitoring, anti-theft protection, automated customs handling, tracking and tracing and more. These applications often integrate the sensor network with existing back-end infrastructure at the transport and logistics company.

1.2 Multi-purpose sensor networks

Early sensor network deployments were heavily optimized for a single application leading to an efficient use of the available energy and thus an increased lifetime of the sensor network. In commercial applications, the lifetime is not the only factor that determines the cost of a sensor network deployment. The cost for application development, the operational expenses for managing the sensor network and the *usefulness* (i.e. the potential for a financial return) of the gathered data also influence the return on investment (ROI) of a wireless sensor network [127].

The potential commercial applications of wireless sensor networks are numerous, even in a single domain, such as transport and logistics (see Section 1.1.3). With this fact in mind, it becomes clear that one sensor network per application is not cost-effective. Such an approach duplicates the hardware cost and the management cost for each application. Sharing and reusing the sensor network for multiple applications avoids this cost duplication. Thus, to increase the return-on-investment (ROI), multi-purpose sensor networks emerge as a new way of using sensor networks. Multi-purpose sensor networks differ from single-purpose wireless sensor networks in two important characteristics:

1. the sensor network is a reusable asset simultaneously hosting multiple applications that evolve over time.
2. the sensor network is used for different purposes simultaneously by multiple actors.

A multi-purpose sensor network is considered an asset that is expected to host multiple applications throughout its lifetime, both sequentially and simultaneously. Consider a sensor network in a trailer that is used to monitor the cargo. The type of cargo varies during the lifetime of the sensor network, perhaps even with each load carried by the trailer. As a consequence, the requirements for monitoring software on the sensor network will vary over time, because the monitoring requirements depend upon the type of cargo. Pharmaceuticals, for example, require an accurate temperature monitoring while humidity is more important for tobacco products.

The driving use cases for multi-purpose sensor networks involve multiple actors interacting with the sensor network. For example, in container logistics both the transport company and its customer—who owns the cargo—are interested in the data that the sensor network gathers. Also customs might be interested in information provided by the sensor network in order to accelerate customs handling. These actors thus expect simultaneous access to different types of information from the same sensor network.

Domains such as industrial automation and mechatronics, traffic and mobility and health care provide other examples of multi-purpose sensor network use cases. Future industrial automation might involve mobile robots guided by information from a sensor network with cameras, proximity sensors and motion detectors. Multiple robots will execute different tasks and require simultaneous access to the information provided by the sensor network. A sensor network deployed throughout a city for traffic and mobility monitoring will host multiple applications such as monitoring of traffic jams, detection of accidents and monitoring of traffic induced pollution. Also in health care sensor networks might be shared for applications such as equipment tracking, patient monitoring and monitoring of environmental conditions in a hospital. These use cases clearly illustrate the variety of operational conditions that multi-purpose sensor networks may exhibit. The network scale and the degree of node mobility and node heterogeneity may vary greatly with the application domain.

1.3 Problem statement

Multi-purpose sensor networks have the potential to increase the return on investment of wireless sensor networks. On the other hand, supporting sharing and reuse of the sensor network introduces extra management complexities. To avoid a substantial increase of the operational expenses, management solutions for multi-purpose sensor networks are needed. This dissertation focuses on management solutions for distributed software applications on such networks.

Potential users of multi-purpose sensor networks expect to manage the sensing infrastructure with the least amount of hassle (similar forces drive the virtualiza-

tion and cloud computing trends for traditional business software). Users want to express their requirements on an abstraction level that matches their mindset. For example, in a logistics context, a deployment request might be expressed as follows: the next load of this trailer contains pharmaceutical products, deploy the necessary software to monitor this load in accordance with the terms in the contract with our customer and in compliance with the current legislation.

State-of-the-art approaches for managing distributed applications in sensor networks provide only low-level operations. They typically provide support for deploying a binary file to a complete sensor network, to a group of nodes (less common) or to a specific sensor node. Most approaches also provide a way to change parameters of a running system or activate and deactivate certain functionality. These approaches target experienced users that require infrequent reconfiguration in single-purpose scenarios.

There is clearly a gap between the expectations of administrators of multi-purpose sensor networks and the current support of sensor network management solutions that appeals to experienced users of single-purpose sensor networks. This dissertation aims to decrease this gap by taking the following three steps:

- Provide autonomous management to relieve the administrator from dealing with the full dynamics and scale of the multi-purpose sensor networks.
- Improve flexibility of the software reconfiguration to accommodate the changing environment and the evolving user-level requirements.
- Provide explicit multi-application support to address the multi-purpose and multi-actor context.

All of this must be achieved with an acceptable resource efficiency to meet the resource constraints of wireless sensor networks. Thus, to conclude:

> The goal of this PhD is to provide resource efficient, autonomous management solutions for flexible deployment and reconfiguration of multiple distributed applications in multi-purpose sensor networks.

As highlighted in Section 1.2, the operational conditions of multi-purpose sensor networks may vary greatly. These operational conditions influence the detailed design of management solutions, however, the work in this dissertation focuses on the common architectural issues in autonomous management, flexible reconfiguration and support for multiple applications in multi-purpose sensor networks. The architectural design accommodates tailoring of the solutions to varying operational conditions such as network scale, degree of dynamism and node heterogeneity.

1.4 Contributions

The contributions in this dissertation address the problem statement in the previous section. These contributions can be summarized as follows:

- Two runtime systems for wireless sensor nodes that provide flexible reconfiguration and the foundation for multi-application support.
- An autonomous management tool for assembly and deployment of multiple distributed applications on multi-purpose sensor networks.

The first runtime system consists of a dynamically adaptable virtual machine for sensor networks, called DAViM (DistriNet Adaptable Virtual Machine). DAViM focuses on providing reconfiguration and multi-application support in the runtime itself through extensive modularization. It leverages upon state-of-the-art techniques for reconfiguration of sensor networks. The second contribution, a component model for multi-purpose sensor networks (LooCI, Loosely coupled Component Infrastructure) extends the modularization to application development time. It provides components as a first-class citizen with support for reconfiguration at runtime and distributed, loosely coupled interactions. DAViM and LooCI are at different ends of the trade-off between more fine-grained reconfiguration on the one hand and cleaner modularization on the other hand.

A management solution for application assembly and deployment (QARI, Quality Aware Reconfiguration Infrastructure) builds on top of these runtime systems. It improves manageability by providing administrators a high-level abstraction to express the goals for the deployment of their applications. QARI autonomously pursues these goals and keeps doing so in the presence of network changes. QARI provides these features for multiple simultaneous applications and manages the trade-off between competing applications. Although both DAViM and LooCI provide reconfigurability and multi-application support, the integrated approach in this dissertation focuses on the integration of QARI and LooCI. QARI benefits from the cleaner modularization of LooCI's component-based approach that eases application assembly and promotes module (i.e. component) reuse.

A prototype implementation, illustrated in the context of transport and logistics, confirms the feasibility of the proposed solutions for resource constrained low-end sensor nodes. The integration of the application management solution (QARI) and the component model for sensor nodes (LooCI) in a testbed confirms the practical applicability of the approach.

The rest of this dissertation is structured as follows. Chapter 2 first details the requirements for the management solutions and describes the approach used to address these requirements. The chapter concludes with an overview of the key

contributions in this dissertation. Chapter 3 presents the contributions in the area of runtime support for reconfiguration. Next, Chapter 4 describes the management solution for application assembly and deployment. Chapter 5 introduces and illustrates the prototype implementations of the proposed solutions and their integration in a testbed. Finally, Chapter 6 lists the conclusions and discusses opportunities for future work.

Chapter 2

Requirements, approach and overview

This dissertation focuses on management solutions for distributed software applications in multi-purpose sensor networks. Section 2.1 derives the requirements for such management solutions from the characteristics of (multi-purpose) wireless sensor networks. Four requirements are identified and discussed: resource efficiency, autonomy, reconfigurability and multi-application support. Section 2.2 discusses literature related to these requirements.

The approach (Section 2.3) followed to address these requirements is threefold. The solutions achieve acceptable resource efficiency by leveraging the resources of the gateway and back-end infrastructure connected to the sensor network. To facilitate autonomous decisions and ease the integration of multiple applications, the management solutions raise the abstraction level of their input policies. Finally, modularization of the sensor nodes' runtime system enables flexible reconfiguration and provides basic mechanisms for multi-application support.

The key contributions (Section 2.4) of the work in this dissertation are (1) two modular runtime systems, DAViM and LooCI, that provide the basic building blocks for flexible reconfiguration and multi-application support and (2) an autonomous management tool, QARI, that manages the assembly and deployment of multiple applications on a multi-purpose sensor network with minimal human intervention. The runtime systems explore the trade-off between more flexible reconfiguration (DAViM) and cleaner modularization (LooCI). Because cleaner modularization benefits application assembly and module reuse, the integrated approach in this dissertation combines QARI with LooCI. This integrated approach is validated through scenarios in a transport and logistics context.

2.1 Requirements

This section discusses the requirements for management solutions for distributed applications in multi-purpose sensor networks. These requirements are derived from the characteristics of wireless sensor networks in Section 1.1.2 and from the characteristics of multi-purpose sensor networks in Section 1.2. Two requirements follow from the characteristics of wireless sensor networks in general: resource efficiency and autonomy.

Resource efficiency: Because the resources are limited, the software on a sensor node should use these resources efficiently. The finite energy supply bounds the life-time of a sensor node because the usage of resources consumes energy. Software on a sensor node should be aware of these constraints. Thus, the parts of management solutions that require processing or communication within the wireless sensor network should be sufficiently lightweight and efficient.

Autonomy: Due to the high-level of dynamism and the large-scale of wireless sensor networks, it becomes infeasible to manually manage all aspects of such network. Software running in wireless sensor networks should therefore have a high degree of self-organization and self-configuration. This also applies to the solutions that manage this software. Management solutions should thus have a sufficient level of autonomy to avoid the need for manual intervention when network conditions change.

Since this dissertation targets the field of multi-purpose sensor networks, the characteristics of this use case of sensor networks must be taken into account. Two additional requirements stem from these characteristics: reconfigurability and multi-application support.

Reconfigurability: The software on a multi-purpose sensor network should be dynamically reconfigurable in order to enable adaptation to new applications or changing environmental conditions. The reconfiguration should be flexible enough to support both fine-grained changes and course grained replacements or additions of functionality. This is the driving requirement for the work in this dissertation. This dissertation aims to provide management solutions that assist a network administrator in the initial configuration and subsequent reconfiguration of distributed applications on a sensor network.

Multi-application support: In order to meet the requirements of the diverse actors, the owner of a sensor network should be able to open its network for third-party applications. In return for direct payment or

reciprocal application hosting, a sensor network owner should be able to allow trusted third parties to deploy custom software to its sensor nodes. A sensor network or a single sensor node should thus be able to run multiple applications—potentially originating from divers actors—concurrently. Consequently, management solutions should be able to simultaneously manage multiple applications on the same sensor network. In the light of the resource efficiency requirement, an approach that encourages re-use of existing functionality in multiple applications is preferred.

2.2 Related work

This section discusses state-of-the-art related to the requirements introduced in Section 2.1. Section 2.2.1 discusses related work on autonomic computing, an approach towards IT systems that autonomously manage their own operation. Section 2.2.2 covers the state-of-the-art in dynamic change management, i.e. the management of runtime changes to a distributed system. It also discusses the impact of the characteristics of wireless sensor networks on dynamic change management. Finally, Section 2.2.3 discusses multi-application support, a challenge that has long been addressed in general purpose computers, but has received little attention in the domain of wireless sensor networks.

2.2.1 Autonomic computing

IBM coined the term autonomic computing [71] to indicate IT systems that manage aspects of their operation without human intervention. The term autonomic refers to the human autonomic nervous system that manages several aspects of the human body without conscious effort of the human.

The self-managing behavior of autonomic computing systems can be divided into four categories: self-configuration, self-optimization, self-healing and self-protection [63]. Self-configuring systems are able to configure themselves based on high-level goals. Self-optimizing systems autonomously optimize their resource use or quality-of-service. When a system autonomously detects, diagnoses and repairs problems it possesses self-healing capabilities. A self-protecting system protects itself from malicious attacks or inadvertent changes by users.

The reference architecture for autonomic computing (see Figure 2.1) suggested by IBM [71] is often called the MAPE-K loop (monitor, analyze, plan, execute, knowledge). An autonomic element in the reference architecture consists of a managed element and an autonomic manager. The term managed element refers to the system that is being controlled. The autonomic manager gathers

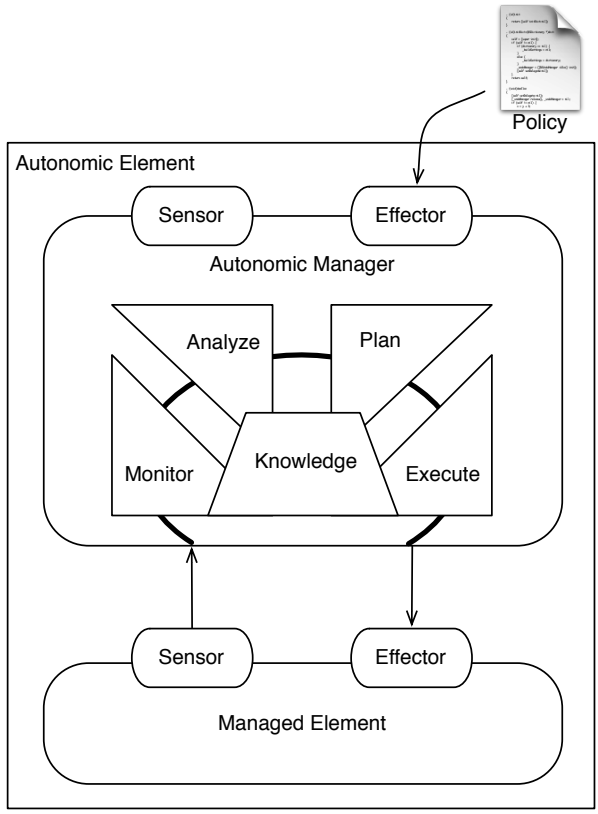


Figure 2.1: The MAPE-K reference architecture for autonomic computing [71]

information about the managed element through sensors and executes changes to it via effectors. The autonomic manager contains the intelligence of the autonomic element.

The Monitor and Analyze building blocks gather and structure information about the managed element. Monitoring collects low-level information about the managed element through the sensors. The Analyze building block performs analysis on this information to detect significant events. For example, it correlates information from different sources or calculates moving averages in order to detect unusual patterns that indicate a problem. The Plan building block handles the planning aspect of autonomic computing. The planning is initiated by a request for planning from the Analyze building block or whenever the policies that govern the autonomic manager change. The planning generates a plan to configure, optimize,

heal or protect the managed element. The Execute building block executes this plan by making sure the necessary changes to the managed element are effected. It uses the interfaces provided by the effectors to achieve this. All the building blocks share information through the Knowledge building block. This is an information repository that is used to store information about the managed element or the autonomic management itself.

An administrator provides policies to govern the behavior of the autonomic manager. An autonomic manager may support several types of policies. Kephart et al. [80] identify action policies, goal policies and utility function policies. Action policies are defined in terms of event-condition-action rules and thus define what to do when the system is in a particular state. Goal policies define either a single desired state of the system or a set of criteria that define the set of acceptable states. Utility function policies define an objective function that allows to rank each possible state. They generalize goal policies and they allow for objective decision-making in case policies compete or conflict.

Autonomic elements could potentially be hierarchically stacked. The autonomic manager then adopts the role of managed element and a higher-level autonomic manager is stacked on top. This enables hierarchical refinement of goals. To facilitate such stacking, an autonomic manager also has sensors and effectors that can be used to respectively monitor the autonomic manager and change the policies in the autonomic manager.

Emergent applications of autonomic computing are situated in the areas of power management, data centers, clusters, grid computing and ubiquitous computing [63]. This dissertation applies autonomic computing techniques to enable autonomy in the management of software applications on multi-purpose sensor networks.

2.2.2 Dynamic change management

Dynamic change management is the management of changes to a distributed software system in a dynamic way, i.e. without stopping the running system. Dynamic change management seeks to provide generic, application independent support for safe dynamic software reconfigurations. It achieves this by providing predefined change functionality that is decoupled from the application to be reconfigured.

Most approaches to dynamic change management are based on component-based software engineering [85, 84, 141, 113, 13, 62] since component-based approaches allow to easily separate structural concerns from application-specific concerns. This facilitates the definition of change scenarios that are independent of the application's functionality. An example of such a scenario is *replace component*.

This expresses a structural change to the software system, but is independent of the functionality implemented by the component. The underlying component framework should implement an API for component control to support the dynamic change management which is built on top. It typically provides functions to add or remove a component, to start or stop a component and to connect or disconnect two components.

The reconfiguration algorithms included in the dynamic change management systems use this component control API to execute changes to the running system. These algorithms sequence the change actions and coordinate the execution of the actions in different parts of the system. Most approaches include a single and fixed algorithm that is able to execute changes in a system independent of the current state of the system.

Preserving the consistency of the system during a reconfiguration is an important requirement for a reconfiguration algorithm. Quiescence [85] is a sufficient condition on the node's state to preserve the system's consistency during reconfiguration. Apart from being a sufficient condition for consistency, quiescence is also guaranteed to be reachable. However, it may take a long time to put the system in this state, causing a significant disruption to the service provided by the system.

Vandewoude et al. [137] showed that quiescence is a sufficient condition, but not a needed condition. Tranquility relaxes quiescence in order to reach a reconfiguration safe state faster. Tranquility is not guaranteed to be reachable but in practice this causes almost never a problem. When it does, the change management system can detect this reachability problem and fall back to quiescence.

NeCoMan [76, 75]—a framework for customizable dynamic reconfiguration in programmable networks—showed that, under certain conditions, reaching a reconfiguration safe state is unnecessary. These conditions depend upon the characteristics of the application and components that are to be reconfigured. Although this renders the reconfiguration algorithm dependent upon the application characteristics, customizing the algorithm based on these conditions is worth the effort because it can speedup reconfiguration and thus shorten the disruption to the application.

One characteristic of wireless sensor networks has an important impact in this context: the reliability of a single node or network link. This reliability is lower in wireless sensor networks than in traditional distributed systems. From this observation it becomes apparent that in wireless sensor networks failures are the norm, not the exception. Of course, failures occur in all distributed systems, but in traditional distributed systems one can often limit the impact of failures by means of timeouts and failure detectors. In wireless sensor networks, this would become troublesome. It is essential for good application design in sensor networks to switch mindsets and deal with these faults by means of fault-tolerant applications.

The unreliability of nodes combined with other factors such as mobility results in much higher dynamics in a sensor network in comparison to traditional distributed systems. The available nodes and the network layout in traditional systems are mostly stable. Changes do occur, but the pace of these changes is lower than in the context of wireless sensor networks. Despite the unreliable nodes and the fast network dynamics, providing reliable services on a wireless sensor network is possible through redundancy [122].

The characteristics discussed above result in loosely coupled interactions between sensor nodes. Loosely coupled interactions are less focused on communicating with specific nodes, but allow switching the interactions to a different node in case of node failures, degraded performance or network disconnection. In the context of dynamic change management, this results in the following benefit: reconfiguration of a node is just another disruption at the node level. An interacting node cannot distinguish between a node that doesn't respond due to failure, or a node that doesn't respond due to an ongoing reconfiguration. If node software is built fault-tolerant, it can deal with both cases. As a consequence, heavyweight algorithms to reach a reconfiguration safe state such as quiescence or tranquility are superfluous. This is an important observation since literature shows that it is difficult to reach such reconfiguration safe states in wireless sensor networks [55].

2.2.3 Multi-application support

The research challenges in the context of multi-application support in general purpose computers have long been addressed by the introduction of time-sharing and multi-tasking in operating systems. In wireless sensor networks, these features are not fully supported by the popular operating systems because early wireless sensor networks were designed and built for a single purpose. Support for these features would put additional burden on the scarce resources while such support is not needed in a single purpose context.

Despite the lack of multi-application support in sensor network operating systems, the need to integrate data and services from wireless sensor networks with multiple back-end applications has become apparent and several approaches have addressed this. For example, distributed database approaches for wireless sensor networks—such as TinyDB [97], DSWare [94], Cougar [146] and SINA [124]—accept queries from various sources. Other approaches such as Janus [40], SONGS [96], GSN [1, 3, 2], SwissQM [106, 108, 107] and the edge server approaches of IBM [120] and Oracle [111] also provide various parties access to the data produced by a sensor network or to the services running on a sensor network. These approaches apply various techniques—Web Services, for example—to expose the sensor network to interested back-end systems.

Recently, the Web of Things (WoT) community has advocated and developed approaches and protocols to integrate ubiquitous computing devices, including sensor devices, into the World Wide Web. The WoT community wishes to make the data and services of these devices universally accessible through the World Wide Web. For example, the CoRE working group of the Internet Engineering Taskforce (IETF) [73] works towards standardization of CoAP, a protocol for constrained RESTful environments that allows easy interoperability with HTTP. This protocol is currently being implemented in, amongst others, the Contiki operating system [41] for wireless sensor devices.

These approaches provides access to sensor network data for multiple applications running outside the sensor network. The characteristics of multi-purpose sensor networks (see Section 1.2) clearly indicate the need to go even further and support multiple applications on the sensor network itself. However, to the best of our knowledge, there is little work in the literature that addresses this challenge.

Federated and shared use are two related though different concepts in the context of security for multi-purpose sensor networks [68]. Federated use means using the sensor infrastructure in different administrative domains to achieve an application goal. In shared use, multiple applications use a single sensor node or network. Enabling shared use implies multi-application support. Efstratiou et al. [43] describe a demonstration of a shared sensor network infrastructure. Rezgui et al. [118] propose the use of a service oriented approach to implement customizable sensor networks that are able to serve multiple applications with distinct needs.

2.3 Approach

This section details the approach followed to address the requirements outlined in Section 2.1. The followed approach is threefold:

- To avoid resource efficiency problems, the proposed solutions leverage the resources available in the gateway and back-end tiers.
- To facilitate autonomy and to ease multi-application support, the presented management solutions raise the abstraction level of the policies that guide their operation.
- To provide reconfigurability and to enable multi-application support, the software on the sensor nodes is modularized.

At first sight, the resource efficiency requirement conflicts with the other three requirements. Although a trade-off between resource efficiency and the other requirements is indeed unavoidable, it should be noted that wireless sensor

networks almost always integrate with gateway and back-end infrastructure. The proposed solutions therefore maximally leverage the resources in the gateway and back-end tier to minimize the impact on the wireless sensor network tier.

Successful autonomous behavior in a management solution depends on the availability of a policy to govern the autonomous decisions. The purpose of these policies is to give the management solution insight into the goals of the system administrators. The higher the abstraction level of the policies, the more insight the policies provide, which allows for better autonomous decisions. In addition, the availability of more information on the administrator's goals eases trade-offs between applications in a multi-application context. Therefore, the solutions in this dissertation raise the abstraction level of the policies provided by the system administrators.

A unit of reconfiguration smaller than the node runtime is key to flexible reconfiguration. It enables independent reconfiguration of these fine-grained modules. Easier assembly of code from multiple applications into a single node runtime follows from that. Therefore, the work in this dissertation modularizes the runtime of the sensor nodes. This dissertation also investigates support for this modularization at application development time because it promotes reuse and eases application assembly.

2.3.1 Leverage resources in gateway and back-end tiers

The challenges brought along with autonomy, reconfigurability and multi-application support seem to imply a negative impact on resource efficiency. This is true to a certain extent: addressing the autonomy, reconfigurability and multi-application requirements will involve a resource efficiency trade-off. It is thus key to address these requirements with an acceptable impact on the resource use. However, autonomy might also improve resource efficiency because administrators cannot always come up with an evenly optimized solution. In addition, a multi-purpose sensor network on its own is of little use. These sensor networks will almost always be integrated with existing back-end systems via gateways. This opens up an interesting opportunity to leverage the computation and communication capabilities of these back-end and gateway systems.

Offloading computationally intensive tasks to the gateway and back-end tiers is especially viable for management related functionality. This functionality is not part of the core features of the sensor network and thus inherently implies overhead. Management decisions based only on statically available information can easily be made in the back-end or gateway tiers. For decisions requiring monitoring information, the monitoring overhead comes into play. However, the back-end often needs the monitored information anyway to allow, for example, human supervision of the (autonomous) management system.

2.3.2 Raising the abstraction level

The input policy for a management solution is key to allow autonomous decisions about the managed system. The policy tells the management solution what to do (action policies), where to evolve towards (goal policies) or what is important (utility policies). The higher the abstraction level of the policies, the better autonomous decisions become possible because more insight is available into the desires of the administrator. This insight also enables the management solution to decide on the potential trade-off between multiple applications. In addition, a higher abstraction level is more convenient for the administrator because it is closer to its mindset.

The management solutions for distributed software applications in this dissertation therefore aim to abstract some of the details of the sensor network. Ideally, an administrator should be able to target an application to *the multi-purpose sensor network* instead of to a specific (set of) sensor node(s). Of course the management system will need guidance from the administrator to select a suitable subset of the sensor network to run the application. The work in this dissertation therefore investigates how to give the management system insight into what the administrator considers good solutions for this node selection problem (i.e. how to express the utility of a solution).

In modern computing one can see several examples of domains that successfully introduced high-level abstractions for complex infrastructure details: the infrastructure or platform is presented as a single service. One can for example develop applications for and deploy applications on *the grid*, a *high-performance cluster* or in *the cloud*.

The following paragraphs highlight two domains—related to the topics in this dissertation—that also introduced such high-level abstractions successfully: software management tools and programming support for wireless sensor networks. Software management tools automate system administration and are thus similar to the solutions for autonomous management of distributed applications in sensor networks that this dissertation aims for. Programming support for wireless sensor networks is complementary to the work in this dissertation: the domain focuses on abstractions for developing applications for wireless sensor networks, while this dissertation focuses on the management of applications at runtime.

Early software management tools provide only thin abstractions over manual management. These tools feature support for automating some of the tasks of software management, for example, through the use of scripts. The abstractions in use are however very close to the physical layout of the managed servers and networks.

A hierarchy of abstraction levels for software management is presented in [38,

36]. At the highest abstraction level, the configuration is expressed as end-to-end requirements such as *configure enough web servers such that the average response time is below X ms.* At the lowest abstraction level, the operations are at bit level. The tools in this lowest category allow, for example, to create a bit-level image of a disk.

PoDIM [37] proposes a policy language at the level of instance distribution rules in this taxonomy. This level of abstraction allows to specify the rules governing the distribution of instances. An example of such rule is *configure N suitable machines as web server.* PoDIM is able to automatically translate these rules to a software configuration.

This dissertation investigates the possibility of raising the abstraction level of software management for multi-purpose sensor networks: it aims to enable deployment of an application to *the sensor network.* Network-level programming abstractions for wireless sensor networks aim to allow programmers to program for *the sensor network.* They provide a programming environment that hides certain details of the sensor network. These systems allow programmers to write applications for a sensor network without worrying too much about the details of the underlying sensor nodes and the network connecting those. There are two broad categories in this type of system: macro-programming and distributed database.

Macro-programming approaches—such as MagnetOS [95], Kairos [56], Regiment [109], RuleCaster [14], Abstract Task Graph [10] and COSMOS [9]—allow a programmer to write programs for the global behavior of the sensor network. These programs are then compiled to node-level behavior that executes on a runtime system.

Another popular abstraction is the distributed database abstraction. Approaches based on this abstraction—such as TinyDB [97], DSWare [94], Cougar [146] and SINA [124]—provide an SQL-like query interface. This interface allows a programmer to specify what data must be extracted from the network. The runtime system then interprets these queries and collects the required information from the network.

2.3.3 Modularization

Modularization of the software running on the sensor nodes enables flexible reconfiguration and provides the foundation for multi-application support. Modularization means that the software is no longer a monolithic block of functionality, but rather consists of a set of software modules. If these modules are retained at runtime, the software on a single node is also a set of binaries instead of a single binary.

After modularization, an application consists of a set of cooperating modules. The relation between this set of application modules and the set of (binary) modules hosted by a node is not a one-on-one mapping. The modules of a single application can be distributed over multiple nodes and a single node can host modules of several applications. In the difference between these two sets of modules lies the foundation for multi-application support.

Modularization also supports the reconfigurability requirement because it leads to more fine-grained, and thus more flexible, reconfiguration. This more fine-grained reconfiguration is also beneficial for the resource efficiency since it requires less code to be transmitted over the network. Since modules are the unit of deployment and reconfiguration, modularization also facilitates independent reconfiguration of different applications which benefits the multi-application support.

Modularization at runtime thus enables flexible runtime reconfigurability and eases multi-application support. However, modularization as such does not provide support at application development time. Component-based software engineering does provide such support at application development time. A software component is defined in [129] as:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

A component-based approach forces developers to explicitly specify the interfaces and dependencies of their software modules. This eases assembly of complex applications and enables reasoning about the structure of those applications. Explicit support for modularization at development time also promotes reuse of modules (i.e. components) by third parties.

2.4 Key contributions

The key contributions in this dissertation are designed to meet the requirements outlined in Section 2.1: resource efficiency, autonomy, reconfiguration and multi-application support. The challenges in addressing these requirements are approached by leveraging gateway and back-end infrastructure, by raising the abstraction level and by modularization (see Section 2.3).

The key contributions are:

- Two modular runtime systems for wireless sensor networks providing the basic building blocks for reconfigurability and multi-application support

(DAViM and LooCI). DAViM and LooCI explore different ends of the trade-off between finer granularity of the reconfiguration on the one hand and cleaner modularization on the other hand.

- An application management solution for autonomous assembly and deployment of multiple distributed software applications (QARI). This solution provides administrators a high-level abstraction to express the goals for their applications and a management tool that autonomously pursues these goals.

Both runtime systems provide reconfigurability and multi-application support although their focus is different. The dynamically adaptable virtual machine (DAViM) focuses on providing flexible dynamic reconfiguration and therefore compromises some aspects of clean modularization. The loosely coupled component-based approach (LooCI) focuses on providing clean modularization while still providing sufficient, though more course-grained, reconfigurability.

Cleaner modularization through a component-based approach eases the management of dynamic changes because it allows to separate structural concerns from application-specific concerns (see Section 2.2.2). In addition, the support for modularization at development time of component-based approaches eases assembly of complex applications and promotes reuse of components by third parties (see Section 2.2.3). Therefore, the proposed application management solution (QARI) integrates with the component-based approach (LooCI). The integrated approach is validated through a prototype of QARI and LooCI in the context of transport and logistics.

The remainder of this section discusses how the different building blocks of the solution interact and where the different parts of the solution are deployed in the infrastructure. Figure 2.2 and Figure 2.3 show a decomposition and a deployment view of the overall approach. The figures show both the modular runtime systems (DAViM and LooCI) and the application management solution (QARI).

Figures 2.2 and 2.3 show applications that consist of interacting modules. These modules are hosted by a modular runtime system, either DAViM or LooCI, that runs on each sensor node. QARI, the solution for application management, runs on the gateway. From the gateway, it provides the assembly and deployment of the applications for the whole sensor network, it thus leverages the gateway resources to ease the burden on the sensor nodes. QARI needs the support of the modular runtime systems to enact changes.

An important difference between DAViM and LooCI on the one hand and QARI on the other hand is the difference in abstraction levels, both in the decomposition view and in the deployment view. DAViM and LooCI modularize the runtime to support flexible reconfiguration and ease multi-application support. Therefore DAViM and LooCI provide a decomposition abstraction at the level of modules.

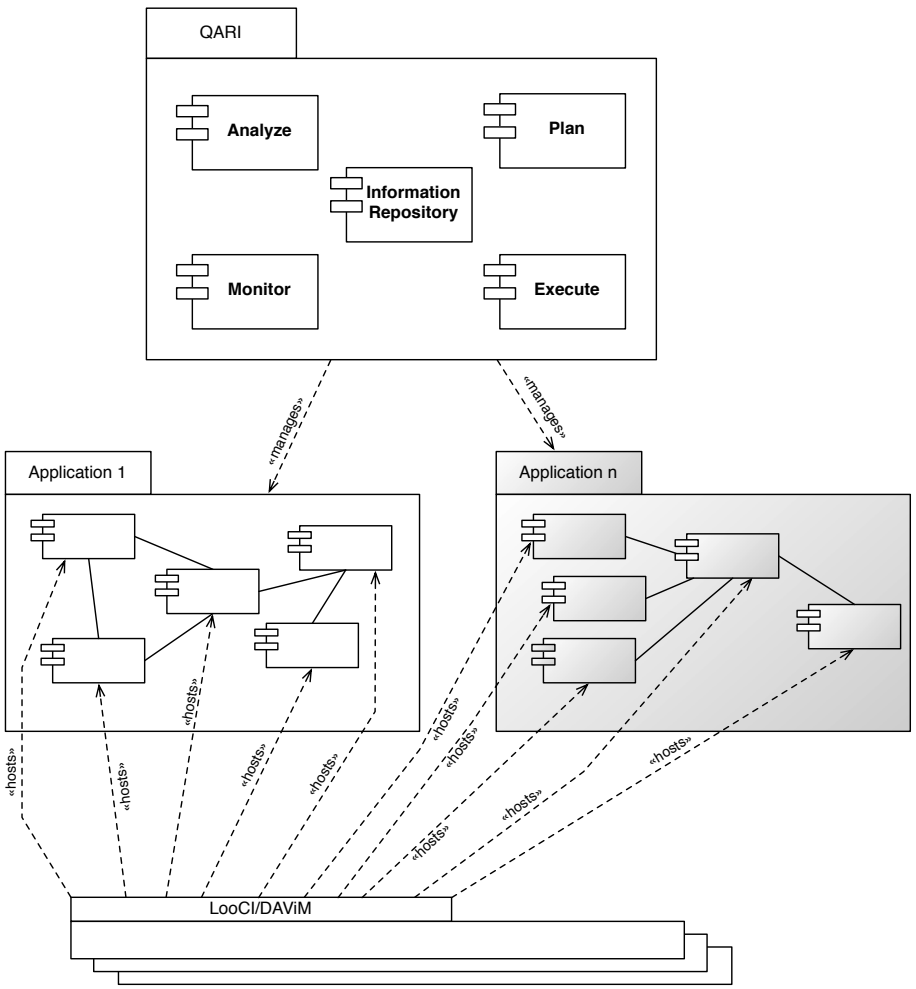


Figure 2.2: Decomposition view of the overall approach.

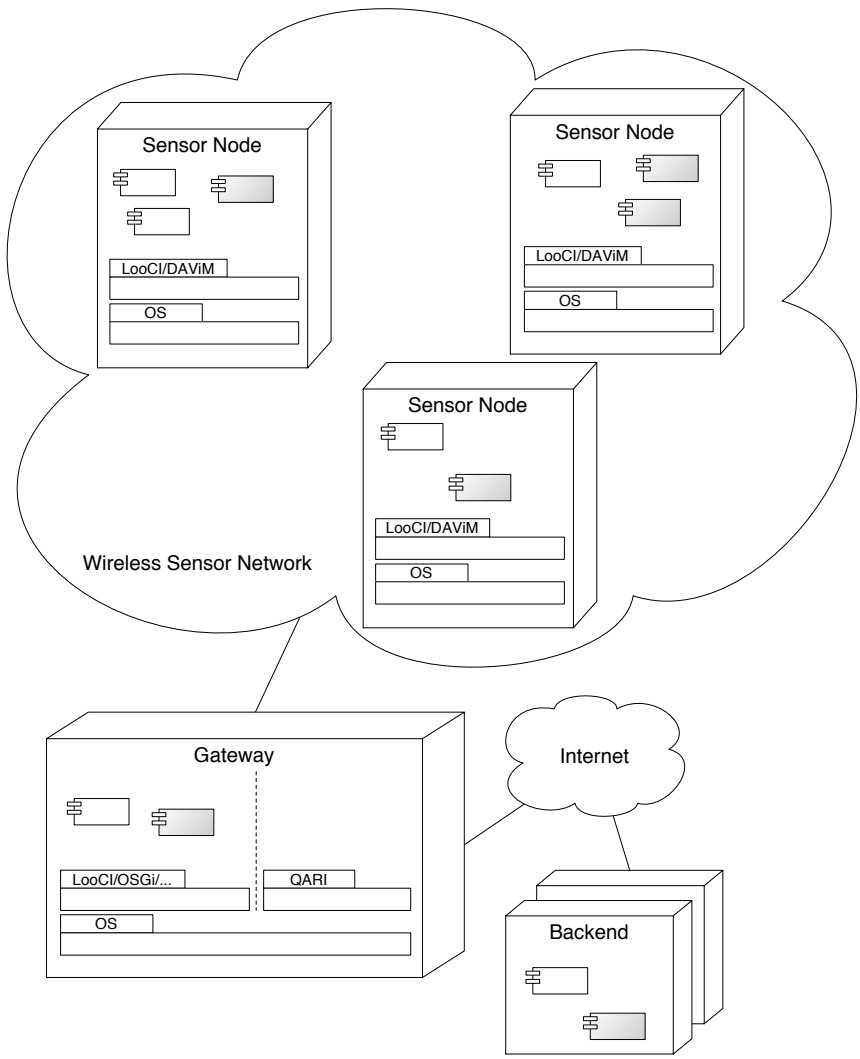


Figure 2.3: Deployment view of the overall approach.

On the other hand, QARI manages applications (assemblies of modules)and thus features application-level abstractions.

The deployment view clearly shows that each node in the sensor network runs a separate instance of either DAViM or LooCI. As a consequence, DAViM and LooCI support node-oriented abstractions in the deployment view. Since QARI manages applications that span multiple nodes, QARI features deployment abstractions at the level of groups of nodes, physical areas and networks.

Thus, the difference in abstraction level between the modular runtime systems (DAViM and LooCI) and the solution for application management (QARI) can be summarized as follows:

- DAViM and LooCI manage modules on a node.
- QARI manages applications on a network.

2.5 Summary

This dissertation presents contributions to the state-of-the-art in management solutions for distributed software applications in multi-purpose sensor networks. These solutions must satisfy the requirements of resource efficiency, autonomy, reconfigurability and multi-application support.

The approach to address these requirements is threefold. First, the solutions leverage the resources of back-end and gateway infrastructure to avoid resource problems in the wireless sensor network tier. Second, the abstraction level of the policies that govern the management solutions is raised to allow for better autonomy and ease multi-application support. Finally, software modularization techniques are used to facilitate flexible reconfiguration and provide the foundation for multi-application support.

The key contributions of the work in this dissertation are (1) two modular runtime systems for sensor networks providing the basic support for reconfigurability and multi-application support (DAViM and LooCI) and (2) an autonomous management tool for assembly and deployment of multiple applications on sensor networks (QARI). DAViM and LooCI explore different ends of the trade-off between fine-grained reconfigurability (DAViM) on the one hand and cleaner modularization (LooCI) on the other hand. Since cleaner modularization provides more benefits to application management, the integrated approach in this thesis focuses on the integration of QARI and LooCI. This integration is validated through a prototype of QARI and LooCI in the context of transport and logistics.

Chapter 3

Runtime support for reconfiguration

This chapter focuses on support for flexible reconfiguration and basic mechanisms for multi-application support. As outlined in Section 2.3, these requirements are addressed through modularization of the sensor node's runtime. Figure 3.1 positions the work in this chapter in the overall approach introduced in Section 2.4. As Figure 3.1 shows, the contributions in this chapter support the management of modules on a single sensor node.

This chapter presents two contributions that approach the modularization of the software at runtime from a different angle. First, an adaptable virtual machine (DAViM) is presented that focuses on maximizing the modularity and reconfigurability of the runtime itself. To achieve this extensive modularity, DAViM leverages state-of-the-art techniques for dynamic reconfiguration of sensor nodes. Second, a component-based approach (LooCI) is described that focuses on providing modularity both at run time and at application development time. LooCI features modularity as a first class citizen to accomplish this goal.

Both approaches have benefits and drawbacks. On the one hand, the component-based approach of LooCI provides explicit module dependencies leading to improved module re-usability and easier assembly. On the other hand, it implies reconfiguration at the granularity of components. In contrast, the adaptable virtual machine approach of DAViM allows for reconfigurations with different granularities. However, DAViM lacks exposure of the modularity at development time leading to opaque and implicit module dependencies that hamper re-use and assembly.

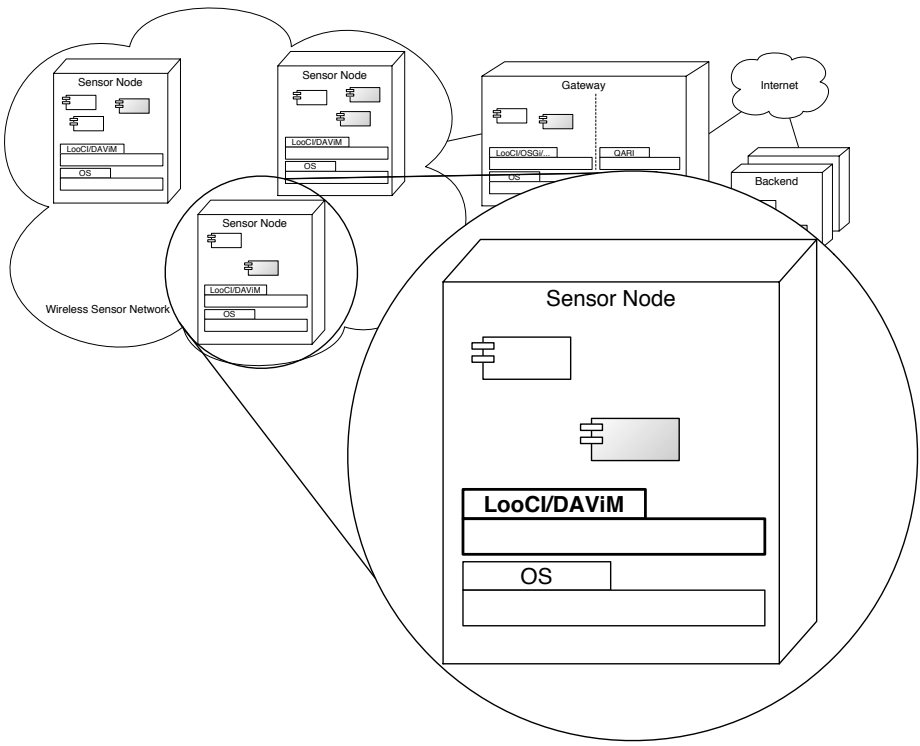


Figure 3.1: Positioning of the work on runtime support for reconfiguration.

3.1 State-of-the-art in dynamic reconfiguration of wireless sensor networks

Section 2.2.2 presents related work on dynamic change management, i.e. the management of dynamic changes to the software running on a distributed system. Such dynamic change management requires mechanisms providing low-level support for dynamic reconfiguration of the software on a node. This section provides an overview of the state-of-the-art in dynamic reconfiguration of the software running on a wireless sensor network.

Dynamic reconfiguration of wireless sensor networks actually consists of two problems: (1) providing runtime support for dynamically loading code and executing that code afterwards and (2) disseminating the code to the appropriate nodes. In theory both problems are orthogonal to each other and solutions for the two problems can be combined at will. In practice, the runtime solution and

the code dissemination technique are often combined in a tightly coupled system. The contributions described in this chapter focus on the runtime support. These contributions use an existing code dissemination approach, however, they can be combined with any type of code dissemination.

Runtime support. The approaches to runtime support for reconfiguration of wireless sensor networks broadly fall into three categories [58]:

- **Monolithic:** these approaches replace all functionality during the update by re-flashing the node's entire program memory with the new code. This requires a reboot of the node and thus results in the loss of all non-persistent state. Some approaches in this category reduce the size of the code updates by only sending the difference between the new system image and the old one.
- **Script-based:** these runtime systems consist of a virtual machine that executes (bytecode) scripts. The bytecode of these virtual machines includes high-level operations when compared to the machine code of the underlying platform. This allows complex applications to be efficiently encoded and thus drastically reduces the size of code updates. This approach stems from the observation that typical sensor network applications use broadly the same building blocks in slightly different ways.
- **Modular:** modular runtime systems provide support for loading coarse-grained pieces of executable code (modules) into a fixed kernel without the need to completely re-flash the node's code memory, nor reboot the node. This typically requires a form of runtime dynamic linking and code relocation on the node.

A monolithic approach is not suited for multi-purpose sensor networks where nodes are expected to execute different tasks throughout their lifetime. In addition multiple concurrent applications may use the nodes. Rebooting the node and thus losing all state with each update is therefore not desirable. The script-based approach can be suited for multi-purpose sensor networks provided that the scripts of different applications (and their state) are isolated. The introduction of such isolation is one of the contributions of this research to the state-of-the-art (see Section 3.2).

TinyOS [61], the most well known and one of the most used operating systems for wireless sensor networks, is a monolithic system. Applications built on TinyOS are compiled together with the operating system into a statically optimized system image. Deluge [66] provides support for dynamically updating TinyOS nodes. Deluge stores multiple system images in external memory and listens for commands

from the network to flash one of the images to the node's program memory and subsequently reboot the node into the new image.

FlexCup [98] extends the TinyOS compilation process with metadata generation which allows it to implement a difference based approach for updating TinyOS nodes. FlexCup only disseminates the changed parts of the system image along with the metadata. On the nodes, it reconstructs the new image using the old image, the difference information and the provided metadata. The reconstruction of the image requires several techniques, such as reference patching, that are also used in the modular approaches.

SOS [57] and Contiki [39] are representative examples of the modular approach. SOS modules are position independent binaries that interact through clean messaging and function interfaces. The use of position independent binaries prevents the need for relocation on the nodes. The function interfaces are registered with the kernel at module initialization time. Contiki includes a (C)ELF-loader that is capable of loading (compact) ELF-binaries. The loading process includes relocation and reference patching on the node.

Sentilla Perk [27] and SunSPOT [112] provide modular approaches based on the Java ME platform. These platforms allow the dynamic deployment of MIDP 1.0 compliant applications. Although these platforms include a Java ME compliant virtual machine, they are categorized as modular platforms. The Java bytecode provides a thin abstraction layer over the hardware rather than a high-level abstraction of widely used functionality that is the basis of the script-based approaches.

Maté [90] pioneered the script-based approach. It features a virtual machine on top of TinyOS that interprets a high-level bytecode. The bytecode includes operations for often used functionality in sensor networks, such as sampling a sensor or sending a packet to a sink node. The use of this high-level interface results in very small application programs that typically fit into a single network packet, which can be disseminated very efficiently. Active sensor networks [91] extends Maté with support for customization of the bytecode instruction set. The customization happens through assembly of the instruction set at compilation time.

Combinations of the above approaches are of course possible. Monolithic techniques can be used to update the kernel of a modular system or the virtual machine in a script-based approach. DVM [11] provides multi-level reconfiguration by combining the modular and the script-based approaches.

Code dissemination. The protocols for disseminating an update in the sensor network differ in several ways, including target selection, reliability features and efficiency. In the context of this dissertation, the most important discriminator is the target selection mechanism. Three possibilities can be distinguished: (1) the

dissemination always delivers the code to the whole network (similar to broadcast), (2) only a single node receives the code (similar to unicast), or (3) the code gets distributed to a group of nodes (similar to multicast). If the protocol disseminates the code to all nodes of the network, the runtime might of course provide support for filtering updates before installation. In the case of group support, an orthogonal issue of group membership management arises.

In the context of multi-purpose sensor networks, not all nodes in the network will run the same software. Support for targeting groups of nodes is therefore highly desirable. It can be emulated by concurrent or sequential unicast deployment, but this will result in a higher overhead on the network for the duplicated transfers of the code to the desired nodes.

In the traditional view of sensor networks all nodes run the same application, therefore the protocols for dissemination to the whole network outnumber the solutions for unicast or group-based dissemination. A comprehensive overview is available in the literature [128]. Trickle [93] and Deluge [66] are the most well-known amongst these protocols. Trickle is used by Maté [90, 91] to disseminate small bytecode scripts to all nodes in the network. It is an epidemic protocol based on polite gossip: nodes only broadcast an update if they didn't already hear a neighbor broadcast it. During the maintenance phase—the period between two updates—the nodes periodically broadcast version information. The frequency of these broadcasts exponentially decreases to avoid high maintenance overhead, but to assure fast updates it resets when Trickle detects an inconsistency. Deluge is based on Trickle but in contrast to Trickle, it supports the reliable dissemination of large code images.

The SunSPOT [112] platform is an example of a platform that uses a unicast connection to a node to send code updates. TinyCubus [99] and FiGaRo [105] feature group-based dissemination. TinyCubus includes a role-based code distribution mechanism based on Generic Role Assignment [119]. The group-based code dissemination mechanism of FiGaRo is inspired by the earlier work of the authors on dynamic grouping of nodes using node properties (Logical Neighborhoods [104, 24]). However, the specific requirements of reliable code dissemination required the design of a new protocol.

3.2 An adaptable virtual machine for sensor networks

The goal of this chapter is to provide flexible reconfiguration and to ease multi-application support through modularization. This section presents an adaptable virtual machine (DAViM) that features extensive modularity through a combination of the script-based approach and the modular approach introduced

in Section 3.1. In addition, DAViM isolates different applications into their own virtual machine and provides dynamic customization of these virtual machines.

DAViM's main features are:

Basic virtual machine support. DAViM uses the script-based approach and thus includes a virtual machine that provides support for running bytecode scripts. DAViM also supports dynamically updating these scripts.

Dynamic deployment of operation libraries. DAViM groups the operations in the instruction set of a virtual machine in operation libraries. DAViM allows such libraries to be added, updated or removed at run-time. It relies on the operating system to provide support for loadable code. Once the code for an operation library is loaded, the library is plugged into DAViM and becomes available for inclusion in the instruction set of a virtual machine.

Support for multiple virtual machines. DAViM supports multiple concurrently executing virtual machines. These virtual machines can be dynamically added, updated or removed. This avoids the difficult task of predicting the number of virtual machines that will be needed during the lifetime of the sensor network. The virtual machines are completely isolated from each other by the DAViM core. The DAViM core also handles the scheduling of the multiple virtual machines.

Flexible composition of virtual-machine instruction sets. Because the size of a virtual machine instruction set is limited, only a subset of the available operation libraries can be included. Depending on the application, the different virtual machines need different operations in their instruction set. Therefore DAViM allows to define the instruction set on a per virtual machine basis by dynamically mapping blocks of byte codes to operation libraries. As a consequence, application byte code is only meaningful if this mapping is also available.

Figure 3.2 shows the architecture of DAViM. The architecture consists of five main building blocks that provide the features above: the Application Store, the VM Controller, the Operation Store, the VM Store and the Coordinator.

Application Store. As part of the basic virtual machine support, the Application Store is responsible for the management of the application components (i.e. bytecode scripts with application functionality). It cooperates with the Coordinator to ensure the application components stay up-to-date. When a new or updated application component arrives (through the Coordinator), the Application Store installs the component in the correct virtual machine. It thus also has awareness of the multiple virtual machine feature.

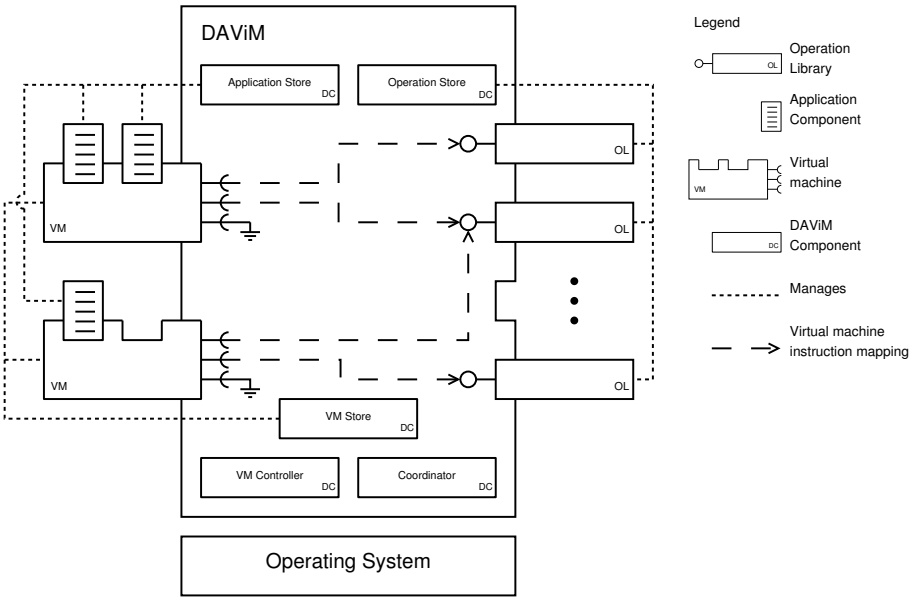


Figure 3.2: High-level overview of the DAViM architecture.

VM Controller. The VM Controller implements the engine that controls the operation of the virtual machines. It schedules the execution of the application components installed in the different virtual machines and routes events to the correct virtual machine. It thus cooperates in the basic virtual machine support and the multiple virtual machines features.

Operation Store. The Operation Store is the main contributor to the support for dynamic deployment of operation libraries. It keeps track of which operation libraries are loaded and cooperates with the operating system to install, update or remove an operation library. When a new operation library becomes available or an existing one is updated or removed, it notifies the VM Store of this change.

VM Store. The VM Store concentrates all information related to the virtual machines. It provides locking of variables to ensure safe concurrent execution of multiple application components in the same virtual machine. It also implements most of the support for multiple virtual machines. In addition, the VM Store is responsible for the dynamic mapping of the virtual machine instruction sets.

Coordinator. The Coordinator handles the dissemination of updates for the basic virtual machine and the multiple virtual machine features. It thus

> disseminates application components and virtual machines in the network. In cooperation with the Application Store, it makes sure all nodes in the network have the latest version of the application components installed. In a similar way, it cooperates with the VM Store to ensure all nodes in the network run the same version of the virtual machines.

To conclude, DAViM provides a modular runtime with fine-grained reconfiguration through its support for application components (scripts), operation libraries and multiple virtual machines. This advanced modularity qualifies DAViM for use in multi-purpose sensor networks that need both flexible reconfiguration capabilities and multi-application support (see Section 5.2.1).

3.3 Supporting modularity at development time

Although the modularity of DAViM is sufficient to address the reconfigurability requirement and to provide basic multi-application support, DAViM's support is limited to the runtime. At application development time, the modularity of the DAViM runtime is not a first class citizen. This introduces a risk for implicit dependencies between modules which hampers application assembly and module reuse across different applications. The implicit dependencies also increase the complexity of managing the software modules' deployment and (re)configuration.

Component-based software engineering (CBSE) and service oriented architecture (SOA) are two approaches from the domain of software engineering that provide better support for modularity at development time. Component-based software engineering provides explicit modularization at development time, independent deployment and third-party composition. Service oriented architecture adds loose coupling and service discovery on top.

The main abstraction in component-based software engineering is a *software component* (often referred to as *component* if it is clear from the context that it concerns software). A software component is defined in [129] as:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Three important characteristics of software components arise from this definition. First, the interfaces explicitly describe the functionality provided by a software component and the receptacles, or required interfaces, make the dependencies for proper operation of a software component explicit. Second, a software component

can be deployed independently. Third, a software component can be composed with other components by a third party, i.e. a party that wasn't involved in the development of the component.

These characteristics make software components a promising abstraction for providing modularity at development time in multi-purpose sensor networks. They facilitate the reuse of components in multiple applications, that may be deployed independently, even by third parties. This potential for reuse is important for multi-purpose sensor networks since sensor network applications will often require common pieces of functionality. At the same time, the resources are too limited to deploy duplicate functionality.

A service in a service oriented architecture resembles a component, but services are loosely coupled and discoverable. Services interact with other services through explicit interfaces, like software components, but services are always loosely coupled to each other. This is achieved through late binding: a service consumer and service provider are bound together as late as possible. In practice, services bind to their dependencies at run-time. The binding can occur once, at startup, or multiple times over the lifetime of the service. In extreme cases, the services use bindings on a per-call basis.

Service discovery makes the functionality that services provide (i.e. their interfaces) and the applicable methods to access the services discoverable by other services. Diverse forms of this service discovery mechanism are possible, ranging from a single centralized service registry to a truly decentralized service discovery protocol. In combination with the loose coupling, this discoverability allows services to search for new services that fulfill their dependencies and subsequently switch the binding for their dependencies to a new provider.

These characteristics of service oriented architecture align nicely with multi-purpose sensor networks and wireless sensor networks in general. The dynamic, unreliable nature of wireless sensor networks will cause bindings between components or services to fail. A service oriented architecture is better suited to deal with such situations than a pure component oriented approach.

Although the runtime can assist the service discovery mechanism by supporting introspection, this is not a prerequisite. The service providers can populate the service registry with static metadata provided by the developers. Therefore, this dissertation focuses on loose coupling which requires support by the component model and the underlying runtime to be effective.

Based on the characteristics discussed above, three criteria are devised to evaluate state-of-the art component-based approaches for wireless sensor networks: (1) run-time reconfigurability, (2) support for distributed interactions and (3) support for loosely coupled interactions.

Runtime reconfigurability. Some component-based approaches only provide configuration support at development time. At compile time, these systems put all components together in a single statically optimized binary for deployment to a sensor node as a single entity. On the other hand, runtime reconfigurable component-based approaches provide support to deploy, remove, start, stop, wire and unwire components at runtime. This implies that component metadata is not optimized away, but remains available at runtime. Note that components in the former type of approach do not comply with the definition of a software component as stated above. The components in such approaches are not independently deployable to a sensor network. Thus, runtime reconfigurability clearly is a necessary characteristic for a component-based approach in multi-purpose sensor networks.

Support for distributed interactions. Various component-based approaches only provide mechanisms for interaction with components on the same node. Components have to rely on the networking support of the underlying platform for distributed interactions. Other component-based approaches also provide an abstraction for distributed interactions. Ideally, this abstraction is the same as the one for local interactions, thus providing transparent local and distributed interactions to components. In a distributed context, which is an inherent property of multi-purpose sensor networks, distributed interactions will occur. Approaches that only support local interactions cannot capture these distributed interactions explicitly, and thus fail the *explicit context dependencies only* characteristic of the software component definition above. Support for distributed interactions is thus essential in multi-purpose sensor networks.

Support for loosely coupled interactions. The interaction paradigm of a component-based approach can be tightly coupled or loosely coupled. A loosely coupled paradigm eases the use of the component-based approach in a service oriented architecture. In addition, a loosely coupled component should continue to operate if one of its dependencies fails. Although it is still possible to write a component that breaks in such situation, a loosely coupled paradigm discourages that and makes doing so harder. This observation has great importance in the context of runtime reconfigurability in a component-based approach. If components don't break when their dependencies become unavailable, it becomes a lot easier or even unnecessary to put the system in a reconfiguration safe state (see Section 2.2.2).

Table 3.1 provides on overview of component-based approaches for wireless sensor networks and indicates whether or not they are runtime reconfigurable, distributed and loosely coupled. NesC [51] was the first component model for WSN. It is the component model used to build the TinyOS [61] operating system. The model only provides support for local interactions and TinyOS provides no runtime

	Runtime reconfigurable	**Distributed**	**Loosely coupled**
NesC	-	-	+
TinyCOPS	-	+	+
Insense	-/+	+	+
Fractal	+	N/A	N/A
OpenCOM	+	N/A	N/A
Gridkit	+	+	-
RUNES	+	+	-
Lorien	+	-	-
REMORA+RemoWare	+	-	+/-
LooCI	+	+	+

Table 3.1: Comparison of component-based approaches for wireless sensor networks

reconfigurability at the component level. TinyCOPS [59] features a component-based approach combined with a content-based publish/subscribe interaction paradigm. The interaction paradigm includes support for distributed interactions. NesC and TinyCOPS both use a loosely coupled interaction paradigm, but in this case they bring no benefit to the reconfiguration since they lack runtime reconfiguration. Insense [33, 12] is a component model based on pi-calculus. Its loosely coupled interaction mechanism includes support for reconfiguring the connections between components. However, it does not provide dynamic deployment or life-cycle support for components.

Fractal [19] and OpenCOM [29] are generic component models that provide abstractions for both components and the connections between components, called connectors. The exact semantics of the connection between two components depends on the specific type of connector used. Gridkit [55] and RUNES [28] provide tightly coupled RPC-style interactions on top of the generic OpenCOM model. Lorien [116] is a sensor node operating system that leverages on OpenCOM to provide unified modularity and reconfiguration at the operating system level.

REMORA [130] supports local event-based interactions between components. Although event-based interaction is essentially a loosely coupled interaction style, REMORA tightens the coupling by imposing the restriction that only one component can produce a certain event type. RemoWare is an add-on to REMORA that provides runtime reconfigurability.

To the best of our knowledge, there exists no component-based approaches for wireless sensor networks that provides all three features: dynamic reconfigurability, a distributed interaction paradigm and support for loosely coupled components

(see Table 3.1). However, the discussion above clearly shows the need for these three features in multi-purpose sensor networks. Therefore, Section 3.4 presents a component model, LooCI, designed with these three requirements in mind.

Although services in a service oriented architecture are often implemented using software components, this is not a requirement. There exist approaches in literature [117, 133, 8, 73] that propose a service oriented architecture for wireless sensor networks without the use of a component model. Others have wrapped sensor networks as services that are offered to interested parties [83, 1, 40, 111].

3.4 A component model for multi-purpose sensor networks

To bring modularity to the application developer, a component model and component runtime for multi-purpose sensor networks, called LooCI, was designed in this thesis. This component-based approach provides runtime reconfigurability, a distributed interaction model and support for loosely coupled components.

3.4.1 The component model

Figure 3.3 illustrates the important concepts in LooCI's component model. A **component** is an independently deployable block of functionality that interacts with other components through well defined interfaces (provided functionality) and receptacles (required functionality). A component has a component type and a component identifier. The **component type** is a free form string that indicates the type of the component. The component developer assigns it at development time. The **component identifier** identifies a component instance within the context of a single node. The component runtime on the node assigns it at deployment time. The separation between component type and component identifier enables multiple instantiations of the same component type.

Components get deployed to nodes that run an implementation of the LooCI runtime. LooCI assumes these nodes have a globally unique **node identifier**. Thus, a specific **component instance** is uniquely identified by the combination of the node identifier and the component identifier.

Components interact by producing and consuming **typed events**. A component declares the event types it produces in its **interfaces**. The **receptacles** of a component declare the event types that the component may consume. A **wire** is a logical connection between an interface and a receptacle. However, components need no awareness of these wires: components always publish events to the distributed event bus or consume events from the distributed event bus.

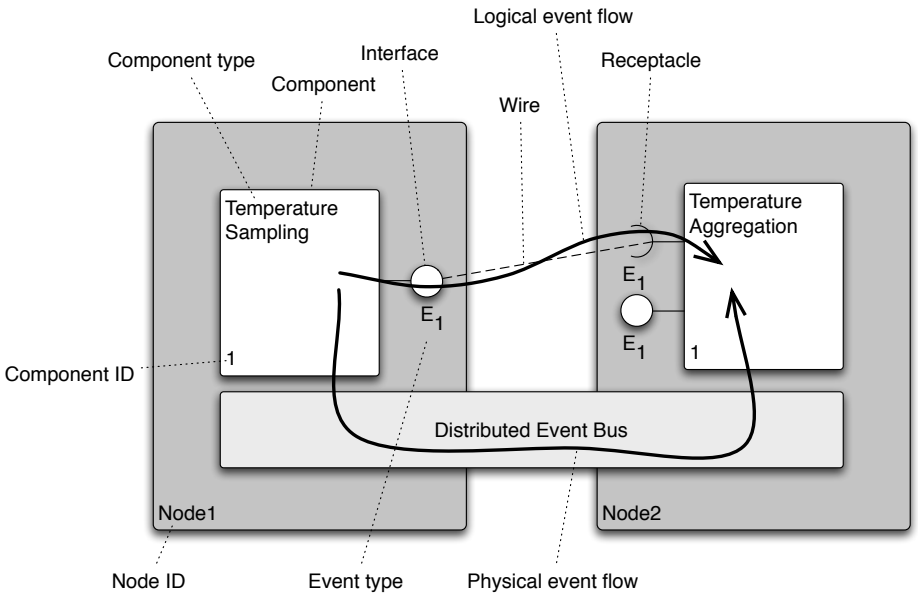


Figure 3.3: Illustration of the important concepts of LooCI, a component model for multi-purpose sensor networks.

The **distributed event bus** implements a decentralized publish/subscribe system. Each wire appears in the distributed event bus as a **subscription** in the subscription table of one or more nodes. For a wire between two components deployed on the same node, the node has an entry in the subscription table of the form $(EventType, ComponentID) \rightarrow ComponentID$. A wire between two components deployed on distinct nodes results in an entry in the subscription tables of both nodes. The entry on the source node has the following form: $(EventType, ComponentID) \rightarrow NodeID$, where $NodeID$ is the node identifier of the destination node. The destination node stores an entry of the following form in its subscription table: $(NodeID, EventType, ComponentID) \rightarrow ComponentID$, where $NodeID$ is the node identifier of the source node. The subscription tables may contain wild cards for node identifiers and component identifiers. If the underlying platform supports group communication, a group identifier may replace a node identifier in the subscription tables.

The **event type** is a data type that allows an *is compatible with* relation between two event types. Currently, event types are bytes and the compatibility relation checks equality. However, future versions of the model will use an event type system that allows a type hierarchy and supports efficient subsumption testing for

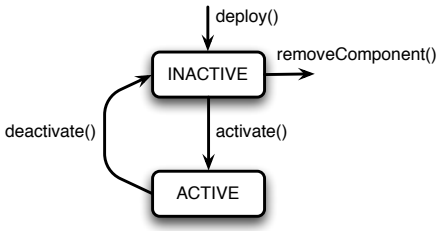


Figure 3.4: The life-cycle of a LooCI component.

testing the compatibility of two event types [131].

3.4.2 Reconfiguration

Runtime reconfigurability constitutes an essential feature for a component model for multi-purpose sensor networks. Since sensor networks are treated as a long-lived, reusable asset, the required functionality changes during the lifetime of the networks. Therefore LooCI's component runtime supports a dynamic component life-cycle.

Figure 3.4 shows the life-cycle of a LooCI component. LooCI supports deployment and removal of components at runtime. A deployed component can be in one of two states: active or inactive. The runtime allows dynamic transition of components between those states.

The life-cycle of LooCI components does not differentiate between deployment and instantiation. This slightly limits the support for multiple instances of the same component type, as enabled by the separation between type and identifier. The lack of separation between deployment and instantiation does not create problems for multiple instances of a component type on multiple nodes. However, in the case of multiple instances on a single node, this leads to duplicated deployment. If the runtime does not detect this duplicated deployment, this will also lead to code duplication on the node. Therefore, future versions of the LooCI component life-cycle will introduce an explicit separation between deployment and instantiation.

LooCI includes a reconfiguration API which consists of three parts: (1) a deployment API to deploy and remove components, (2) a runtime control API to activate, deactivate and wire components, and (3) an introspection API to retrieve information about the current state of the network.

```
1 ComponentID deploy(String filename, NodeID destination);
2 boolean removeComponent(ComponentID component, NodeID node);
```

Listing 3.1: The deployment API of LooCI.

```
1  boolean activate(ComponentID component, NodeID node);
2  boolean deactivate(ComponentID component, NodeID node);
3  boolean resetWires(ComponentID component, NodeID node);
4  boolean wireLocal(EventType interfaceEvent, ComponentID
       srcComponent, EventType receptacleEvent, ComponentID
       destComponent, NodeID node);
5  boolean wireFrom(EventType interfaceEvent, ComponentID
       srcComponent, NodeID srcNode, EventType receptacleEvent,
       ComponentID destComponent, NodeID destNode);
6  boolean wireTo(EventType interfaceEvent, ComponentID
       srcComponent, NodeID srcNode, NodeID destNode);
7  boolean wireFromAll(EventType interfaceEvent, EventType
       receptacleEvent, ComponentID destComponent, NodeID
       destNode);
8  boolean wireToAll(EventType interfaceEvent, ComponentID
       srcComponent, NodeID srcNode);
9  boolean unwireLocal(EventType interfaceEvent, ComponentID
       srcComponent, EventType receptacleEvent, ComponentID
       destComponent, NodeID node);
10 boolean unwireFrom(EventType interfaceEvent, ComponentID
       srcComponent, NodeID srcNode, EventType receptacleEvent,
       ComponentID destComponent, NodeID destNode);
11 boolean unwireTo(EventType interfaceEvent, ComponentID
       srcComponent, NodeID srcNode, NodeID destNode);
12 boolean unwireFromAll(EventType interfaceEvent, EventType
       receptacleEvent, ComponentID destComponent, NodeID
       destNode);
13 boolean unwireToAll(EventType interfaceEvent, ComponentID
       srcComponent, NodeID srcNode);
```

Listing 3.2: The runtime control API of LooCI.

Deployment API. The deployment API (Listing 3.1) handles the deployment and removal of components. It is the only part of the API which is not available from within the network. The `deploy` function (line 1) deploys a given component to a given node and returns the component identifier that was assigned by the node during the deployment. The `removeComponent` function (line 2) removes a specific component instance uniquely identified by the combination of a component identifier and a node identifier.

```
1 List<ComponentID> getComponentIDs(NodeID node);
2 List<ComponentID> getComponentIDsOfType(String componentType,
      NodeID node);
3 String getComponentType(ComponentID component, NodeID node);
4 State getState(ComponentID component, NodeID node);
5 List<EventType> getInterfaces(ComponentID component, NodeID
      node);
6 List<EventType> getReceptacles(ComponentID component, NodeID
      node);
7 List<ComponentID> getLocalWires(EventType interface,
      ComponentID component, NodeID node);
8 List<NodeID> getOutgoingRemoteWires(EventType interface,
      ComponentID component, NodeID node);
9 List<{NodeID,ComponentID}> getIncomingRemoteWires(EventType
      receptacle, ComponentID component, NodeID node);
```

Listing 3.3: The introspection API of LooCI.

Runtime control API. The runtime control API (Listing 3.2) contains support for activation and deactivation (lines 1–2). Together with the deployment API, these two functions allow total control over the life-cycle of a component. The second block of functionality provided by the runtime control API is the management of wires. It allows to create new wires (the `wire*` functions, lines 4–8) and to remove existing ones (the `unwire*` functions, lines 9–13). The `resetWires` function on line 3 makes it possible to remove all wires of a specific component instance at once.

The API supports creation of both local and remote wires. A local wire is a wire between two components on the same node, while a remote wire connects two components that reside on different nodes. A component doesn't notice any difference between these types of wires (see Section 3.4.3), but for the entity that manages the wires this distinction is relevant.

With the runtime control API, one needs to create the two ends of a remote wire separately. A higher level API could be created on top of the runtime control API that provides a single method to wire two remote components. This would improve the usability of the API, but it would also remove some of the flexibility that the current API provides for creating divers distributed interactions (see Section 3.4.3). Part of this flexibility stems from the fact that the wire and unwire methods allow wildcards for the component identifiers and node identifiers. Not all combinations of wildcards are useful or allowed; the `wireFromAll` and `wireToAll` functions (lines 7 and 8) implement two common cases: accepting events regardless of the source and broadcasting events to all interested parties.

Introspection API. The introspection API (Listing 3.3) enables a management entity to discover the state of the network at runtime. An API user can determine the identifiers and types of the components deployed on a given node (lines 1–3). The API also allows to check whether a specific component instance is active or inactive (`getState` on line 4). If the component type by itself does not suffices to assess the functionality of a component, the list of interfaces and receptacles (lines 5 and 6) can be introspected. Finally, the API allows to retrieve an overview of the local and remote wires (lines 7–9).

3.4.3 Distribution

To enable components with explicit context dependencies only, the component model must include a distributed interaction paradigm. When the model lacks such paradigm, components can only interact with remote entities using the mechanisms of the underlying operating system. This introduces implicit dependencies not captured by the component model. Implicit dependencies can cause problems for reconfiguration since the reconfiguration support does not take them into account.

LooCI includes seamless distributed interactions. Components need no awareness of the distributed nature of their interactions. This seamless nature of the distributed interactions is possible because LooCI's interaction model decouples components both in space and time. With tightly coupled interactions, achieving such seamlessness is harder because the latency of distributed interactions will always differ from the latency of local interactions.

The choice for an event-based model facilitates decoupling in time through the use of asynchronous event publication and subscription. Relaying all interactions through a distributed event bus enables decoupling in space because components only need a direct dependency to the event bus. Thus, components only have direct asynchronous interaction with the local event bus. The event bus manages the details of the wires (subscriptions) on behalf of the components.

LooCI's distributed interaction paradigm allows diverse interaction styles. The simplest form of interaction is a one to one interaction. A one to one wiring can be created by the combination of a `wireTo` and a `wireFrom` action with specific arguments for the component identifiers and node identifiers. One to many, many to one and many to many interactions can be setup by using wildcards for the component or node identifiers. If the underlying platform supports group communication, the component runtime can also allow group identifiers as a substitute for node identifiers. Thus, any granularity of interaction can be achieved by using the same two calls to the runtime control API.

3.4.4 Loose coupling

A key feature of LooCI is its loose coupling between components. Thanks to the event based interaction paradigm, a component needs no awareness of its wires. A component publishes or receives events through the distributed event bus. The event bus takes care of the wirings of components. This mechanism decouples components in both space (local or remote wires are transparent to the components) and time (publishing or receiving an event is non-blocking).

The loose coupling encourages component developers to write components that remain operational when dependent components become unavailable. This matches the operating conditions of multi-purpose sensor networks: wireless sensor networks have unreliable communication channels and nodes may fail due to, for example, battery depletion.

If components remain operational when dependent components become unavailable, they also keep functioning when a reconfiguration process deactivates a dependent component (e.g. to update it) or when the component is rewired. The key observation is that a component cannot distinguish between a random failure and a random reconfiguration. Thus, fault-tolerant applications are also reconfiguration-tolerant. This is an important observation in the context of this dissertation (see also Section 2.2.2): it eliminates the need for distributed quiescence [85] or tranquility [137]. These so-called reconfiguration safe states are hard to achieve in unreliable environments such as sensor networks [55].

Apart from the technical aspect of loose coupling, a change in mindset is needed too: a component developer should not rely on the availability of (a specific instance of) another component. The developer should program components that tolerate the failure or unavailability of a dependency and are capable to switch to a new instance of a dependency at any time. This change in mindset cannot be enforced by technical means, but good support for loosely coupled interactions provides an important incentive to developers.

3.5 Discussion

This chapter aims to provide reconfigurability and multi-application support at the level of the sensor node runtime through modularization of this runtime. DAViM and LooCI are two contributions to the state-of-the-art in this context. Although DAViM and LooCI address the same requirements (reconfigurability and multi-application support) through the same approach (modularity), there is an important difference in the focus of DAViM and LooCI:

- DAViM focuses on providing the most flexible reconfiguration through

advanced modularity. DAViM leverages and extends state-of-the-art techniques in sensor network runtime systems to achieve this reconfigurability.

- LooCI focuses on extending the modularity to application development time by making it a first class citizen. LooCI leverages and extends state-of-the-art component-based techniques that have been adapted to the wireless sensor network context.

DAViM achieves more flexible reconfiguration by combining two state-of-the-art approaches for reconfigurable runtime support in wireless sensor networks: the script-based approach and the modular approach. Support for multiple applications is provided through the support for multiple lightweight virtual machines. This results in very flexible reconfiguration that allows an administrator to dynamically update the software on a sensor node at various levels: application scripts, operation libraries and virtual machine configurations.

LooCI leverages on component-based approaches to provide runtime reconfigurable components that connect through a distributed, loosely coupled interaction paradigm. Component-based approaches improve modularity by providing components as a first class citizen. This improved modularity also provides significant benefits for re-usability, sharing and adaptability. Therefore these approaches have been adapted from traditional distributed systems to the wireless sensor network context. LooCI leverages this work to address the reconfigurability and multi-application challenges.

Both DAViM and LooCI have benefits and drawbacks. DAViM provides very lightweight reconfiguration for the cases that can be addressed with an updated application script or with an updated virtual machine configuration. LooCI's reconfiguration is more coarse grained, similar in granularity to the reconfiguration of operation libraries in DAViM. However, the lack of exposure for DAViM's modularity at application development time may result in opaque and implicit dependencies between modules of an application. LooCI does provides exposure of modularity at development time. This exposure leads to easier application assembly, easier management of module (component) dependencies and better re-usability. These benefits over DAViM qualify LooCI as the better candidate to support the application management solution in Chapter 4.

The distinctive features of DAViM in comparison to LooCI are lightweight reconfiguration through bytecode scripts and isolation of multiple applications. LooCI, on the other hand, has explicit modularization at development time that DAViM lacks. If a combination of these features is needed, one could extend LooCI with features from DAViM or vice-versa. Given the fact that LooCI's core features are more beneficial to application management than DAViM's core features, augmenting LooCI with features from DAViM (finer-grained reconfiguration and isolation of multiple applications) is the most promising direction.

A promising approach to enable lightweight reconfiguration in LooCI is augmenting LooCI with policy-based reconfiguration [100]. Policy-based reconfiguration is a technique complementary to component-based reconfiguration that enables customization of a component-based system through declarative policies. These lightweight policies are interpreted at runtime in a way similar to the way DAViM interprets bytecode scripts.

Runtime isolation of multiple applications in LooCI means isolation of the components of different applications. DAViM is able to provide such isolation for bytecode scripts thanks to the interpreted nature of these bytecode scripts. Since LooCI components are based on native code, hardware support such as memory protection and supervisor mode will be needed to enable component isolation in LooCI.

Both DAViM and LooCI provide support for dynamic reconfiguration. An often raised critique against dynamic reconfiguration in wireless sensor networks is the lack of static optimization possibilities. However, it has been shown in literature that this has only a limited effect on the long-term energy consumption of the sensor networks [57]. In multi-purpose sensor networks, where frequent updates are expected, the flexibility of the reconfiguration and the update cost gain importance. Dynamic approaches provide better support in that respect.

3.6 Summary

This chapter described the work to address two requirements put forward in Chapter 2: providing flexible reconfigurability and enabling multi-application support. This chapter approaches these challenges through modularization of the runtime system running on the sensor nodes.

Section 3.2 presented an adaptable virtual machine (DAViM) that improves the runtime reconfigurability of state-of-the-art systems. In addition, DAViM isolates applications in their own virtual machine. DAViM improves the flexibility of the reconfiguration support in state-of-the-art sensor network virtual machines by additionally providing (1) support for dynamically loaded operation libraries and (2) facilitating the flexible composition of the virtual machine instruction set from the operations available in the operation libraries.

Extending the modularity to application development time leads to explicit module dependencies and improved module re-usability. This thesis therefore studied the use of techniques from component-based software engineering and service oriented architectures to achieve such improved modularity in wireless sensor networks. Section 3.3 identifies three essential features for component-based approaches in

the context of multi-purpose sensor networks: runtime reconfigurability, support for distributed interactions and support for loosely coupled interactions.

The study of state-of-the-art component-based approaches for wireless sensor networks in Section 3.3 showed that none of the existing solutions fully supports these three features. This thesis therefore designed a component model (LooCI) that provides extensive support for runtime reconfigurability and offers a distributed, loosely coupled interaction paradigm. This interaction paradigm is an event-based publish/subscribe mechanism that decouples components in both space and time.

Chapter 4

Application management

This chapter presents a management solution for assembly and deployment of distributed applications on top of the component-based runtime system in Chapter 3. The solution improves manageability by supporting a high-level abstraction to express the goals and desires of administrators and by creating a management tool that autonomously pursues these goals. This management tool, QARI, operates on a gateway device at the edge of the sensor network (see Figure 4.1).

The component-based runtime system (LooCI) in Chapter 3 provides mechanisms for deploying and reconfiguring application components on sensor nodes. The management solution in this chapter, called QARI, provides application assembly, application deployment and application reconfiguration on top of these mechanisms. Supporting application assembly through composition of components is a minor contribution, but an essential enabler for the main contribution. The main contribution of the management solution in this chapter is the autonomous deployment and reconfiguration of multiple applications on a sensor network based on high-level deployment instructions.

The high-level architecture of QARI is inspired by the MAPE-K reference architecture discussed in Chapter 2. QARI acts as an autonomic manager for the sensor nodes in a multi-purpose sensor network. QARI's decisions are governed by high-level policies provided by human operators or other management tools.

Section 4.1 first classifies related work according to the abstraction-level and according to the degree of automation. Section 4.2 then shows how QARI's policies raise the abstraction level of the instructions that govern the planning. Next, Section 4.3 illustrates QARI's multi-application support. Section 4.4 continues with details of QARI's architecture, followed by further discussion in Section 4.5.

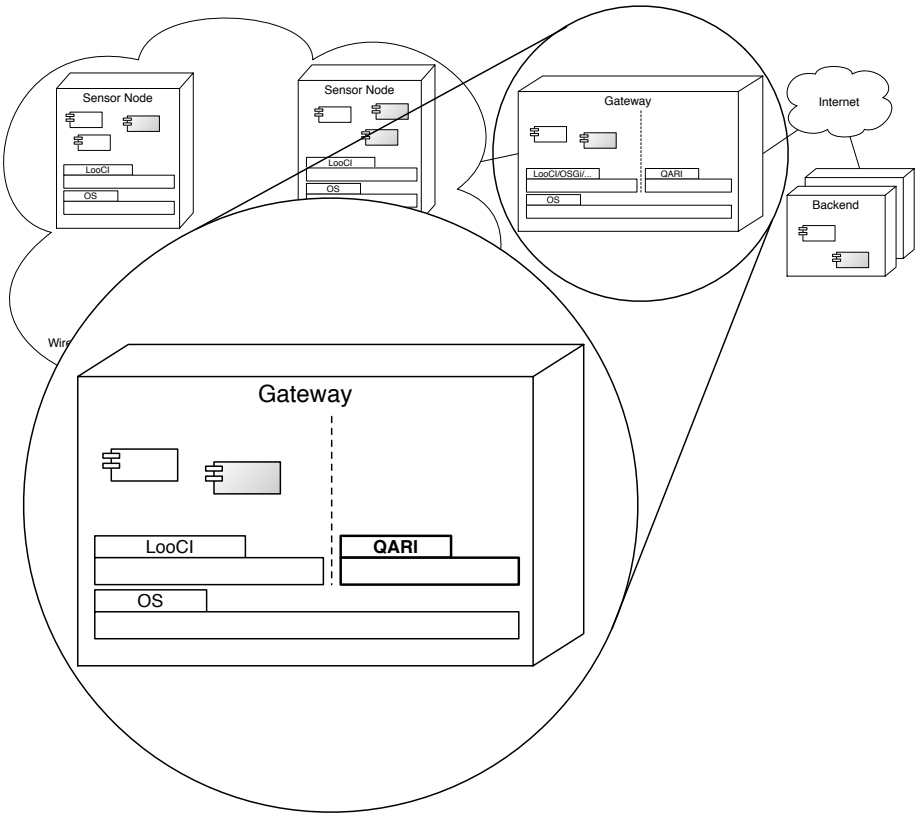


Figure 4.1: Positioning of the application management solution (QARI).

4.1 Related work

This section presents related work in the area of automated management tools for distributed applications. Section 4.1.1 presents a classification of management tools according to their abstraction level and their degree of automation. The approach in this dissertation to provide autonomous management (the highest degree of automation) includes raising the abstraction level. The highest level of abstraction in the classification is based on the utility of a network state (i.e. a value that indicates to what degree a network state satisfies the goals). The aim is to raise the abstraction to this level. Therefore, Section 4.1.2 discusses work in the area of utility in wireless sensor network.

4.1.1 Abstraction level and degree of automation

This chapter presents an autonomous management solution for distributed applications in multi-purpose sensor networks. Autonomy is facilitated by raising the abstraction level of the policies that govern the management tool. Providing autonomy also implies a high degree of automation in the management tool. Therefore, this section classifies related work according to the abstraction level and according to the degree of automation.

The first dimension in the classification discriminates between different abstraction levels for the policies that govern the management tool. The higher the abstraction level of the policies, the more insight the tool has into what the user really wants, which results in more degrees of freedom for the management tool. At the same time, a higher level of abstraction accommodates the user because it is closer to his mindset. The related work is classified according to four levels of abstraction, inspired by the three policy types identified in [80]: (1) action policies, (2) goal policies and (3) utility policies. Within the goal policies the classification in this section distinguishes between policies that describe a single desired state and polices that describe criteria for the system state. The related work is thus classified according to the following four abstraction levels (in ascending order):

Actions: The administrator instructs the management tool in an imperative way. An action policy consists of a sequence of actions needed to bring the system from the current state to a new state. The tool executes these actions on the running system without knowledge of the desired end state. If autonomous behavior is needed, this must be explicitly covered in the policy, through for example specific error-correcting actions or through event-condition-action rules.

Desired state: This kind of policy declaratively describes the desired state of the system. The deployment tool plans a sequence of actions to reach this end state. The tool can autonomously react to the failure of an action by searching for another sequence of actions to reach the desired state. If an event renders the desired state unreachable, the tool requires explicit guidance from the user to resolve the issue (i.e. the user needs to provide another desired state).

Criteria: Policies with criteria no longer describe the desired state of the system explicitly, but rather specify criteria that the system state must satisfy. These criteria define a set of possible target states. The tool first decides which state it will target and then decides on the sequence of actions towards that state. The knowledge of the criteria enables the tool to autonomously react to events that render its target state unreachable. In such cases, the tool switches its target state to another state that satisfies the criteria. The tool can use system-level utility to guide its choice of a target state.

Utility: A utility policy contains enough information to assign a score to each state of the system, either through an explicit utility function or by specifying parameters for generic utility functions provided by the deployment tool. The modus operandi of the tool is similar to that of a tool with criteria based policies: first determine the desired state, then plan a transition towards that state. However, in this case, the tool can base its decision for the desired state on both system-level utility and user-level utility. If the policies originate from different sources, utility provides objective means to decide on the trade-off between different goals in order to reach a system-wide optimum.

The second dimension in the classification, the degree of automation, indicates whether a management tool provides (1) mere automation of deployment, (2) automated deployment and monitoring or (3) fully autonomous deployment, i.e. automated deployment and monitoring and reaction upon the monitoring. This section thus classifies related work according to these three degrees of automation:

Automated: An automated deployment tool realizes the deployment of software based on instructions given by the user. Depending on the abstraction level of these instructions, this includes only execution (actions) or both planning and execution (higher levels). At this degree of automation, the management tool deploys automatically, but doesn't monitor the target system afterwards. Consequently, the only feedback available is feedback on the selection of the desired state and/or the execution of the actions taken towards the desired state. After the deployment finishes, the tool exits or idles until new instructions are given. This type of tool offers already a huge step forward compared to manual deployment.

Monitored: Monitoring the deployed system after the deployment finishes is an important step towards a fully autonomous tool. A tool with this degree of automation does not act upon the monitored information, but merely presents it to the administrator to inform the administrator's decisions. These human-in-the-loop tools allow administrators to react faster to changing conditions because the information for decision making is readily available.

Autonomous: A fully autonomous tool uses the monitored information to react to changing conditions without human intervention. These tools automatically pursue the goals specified by an administrator in a policy. Although it is feasible to build an autonomous tool that reacts to a fixed set of events using a library of actions, autonomous tools typically accept criteria-based or utility-based policies. An autonomous tool might still require human intervention in some cases, for example when meeting the goals becomes infeasible given the changed operating conditions.

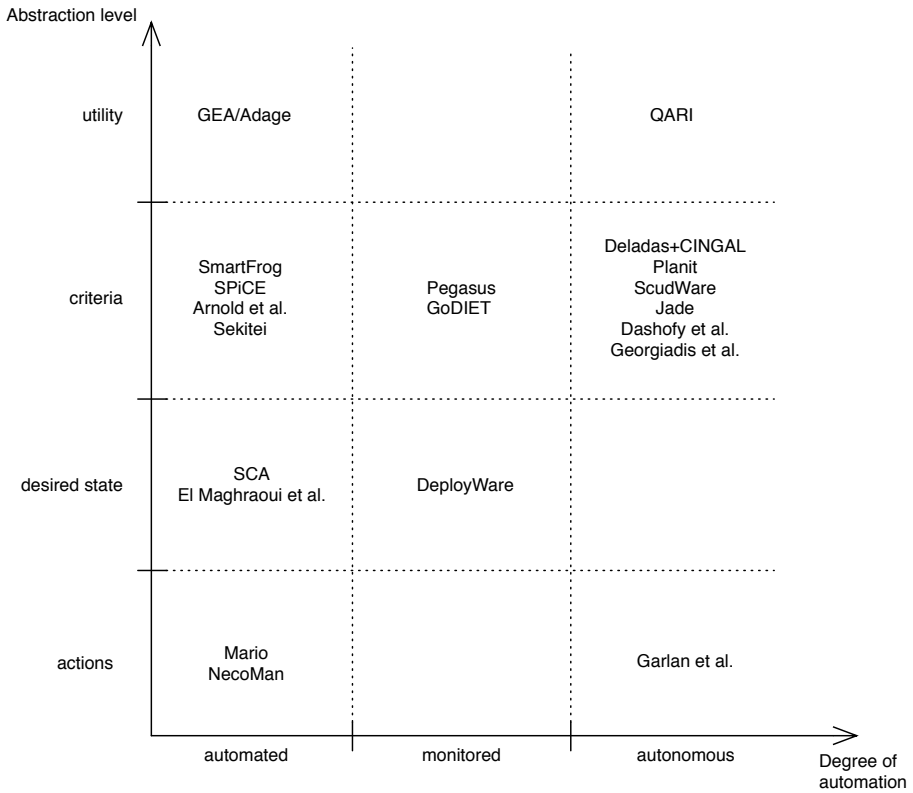


Figure 4.2: Overview of the level of abstraction and the degree of automation in state-of-the-art management tools for distributed applications.

Figure 4.2 shows an overview of the level of abstraction and the degree of automation in state-of-the-art management tools for distributed applications. The x-axis shows the degree of automation and the y-axis depicts the abstraction level. The effort required by a human operator decreases from the lower left corner of the figure to the upper right corner. By going up, the abstraction level rises and thus the effort to provide the input for the tool decreases because the tool takes over the task of transforming utility or criteria into a desired state, or a desired state into actions. By going to the right of the figure, the effort decreases because the tool relieves the human operator of assessing the current state (monitored) and reacting upon significant events (autonomous). The following paragraphs introduce the approaches in Figure 4.2. The text navigates through the figure from left to right and from bottom to top.

Mario [17] automatically deploys flow-based applications to heterogeneous platforms. It uses a common platform-independent component model that includes assembly and deployment instructions. The platform independent assembly instructions specify semantic constraints on component connections. The platform-specific deployment instructions describe how to instantiate, configure and connect the flows.

NeCoMan [76] automatically executes changes in pipe-and-filter based programmable networks. NeCoMan expects a reconfiguration description, a list of all affected nodes, some relevant characteristics of the service involved and the reconfiguration semantics. Based on this information, NeCoMan generates a reconfiguration script that is guaranteed to perform the reconfiguration in a safe way. The reconfiguration description contains an unordered list of actions to execute. NeCoMan reorders the actions on this list to guarantee the safety of the reconfiguration.

Garlan et al. [50] present a model-based approach for self-healing systems. The tool repairs the running system in response to monitored events and is thus fully autonomous. The tool repairs the system using repair scripts which contain actions to be executed whenever a certain condition occurs. The use of scripts for repairing the system limits the type of events that the tool can handle autonomously. When no script is available for a certain event, the tool cannot repair the running system.

The Service Component Architecture (SCA) Assembly specification [110] is a standard for describing service compositions. The format supports composition of components, but does not feature support for describing deployment. The deployment of an SCA assembly depends on the implementation of the SCA runtime. An SCA runtime typically supports the automatic deployment of SCA assemblies, but a description of the desired state is required.

El Maghraoui et al. [47] present a model driven provisioning approach that bridges the gap between a declarative object model of a deployment and procedural provisioning tools. The authors describe an overall approach that starts from an abstraction level with criteria, however in this paper, the focus is on the step after the desired state has been resolved. The output of the preceding phase declaratively describes the desired state, while existing provisioning tools are procedural. The authors present a system that leverages on AI planning to generate an executable procedural deployment plan from a declarative description of the target state.

DeployWare [49] defines a meta-model that captures the abstract concepts of the deployment of a distributed application on a grid. Based on this meta-model, various stakeholders (software experts, system administrators and end-users) describe the desired state of a deployment. The DeployWare runtime, called Fractal Deployment Framework, executes DeployWare descriptions, resulting in a deployed system. The DeployWare eXplorer console can be used to monitor the

state of the system.

SmartFrog [54] features configuration descriptions for components. It describes dependencies on the underlying system software, but these dependencies do not reference specific nodes. The allocation to specific nodes is done at deployment time. SmartFrog automatically validates and executes the software configuration descriptions provided by system administrators.

SPiCE [46, 45] generates distributed deployment models using model transformation techniques. These transformations are performed on input models and according to transformation rules provided by several stakeholders. Developers capture the logical structure of the application, experts describe best practices using deployment model transformation rules, deployers provide the required deployment patterns in a logical deployment model and finally, operators describe the data center resources. A logical deployment model may describe a fixed desired state, it may select specific servers, or it may include criteria for the allowed states in the form of assignments to high-level targets, such as tiers or clusters.

Arnold et al. [4] describe an approach similar to SPiCE, but targeted at service oriented architectures. It uses patterns instead of model transformations. In [5], the authors present algorithms to automatically realize these patterns.

Sekitei [81, 82] provides a planning approach for the component placement problem (CPP), which the authors define as *the placement of application components onto computational, data, and network resources across a wide-area environment subject to a variety of qualitative and quantitative constraints.* Sekitei is aware of resource usage and tries to optimize it. The information provided to sekitei contains no utility preventing sekitei from optimizing application-level quality metrics. Sekitei is limited to planning and does not include an execution engine.

Pegasus [15, 35] uses AI planning for workflow generation in grid computing. It generates workflows based on specifications of the desired data products provided by the user. Pegasus monitors the infrastructure to inform the planning of newly submitted workflow generation requests. However, existing workflows are not re-planned based on this monitoring information.

GoDIET [21, 20] is a deployment tool for the DIET toolkit for grid computing. GoDIET defines a good mapping of software components to resources as one that maximizes the steady-state throughput of the system. After the deployment, GoDIET can collect monitoring information through the use of a LogService that collects traces from all components.

Deladas [101] is a declarative language, similar to an architectural description language, that allows administrators to describe criteria for the desired state of a distributed application. The Deladas runtime compiles this description into a constraint model. The solutions of this constraint model are mapped back to the

problem domain to yield the possible configurations. The system then chooses a configuration and deploys it through CINGAL [34]. To allow autonomy, the approach also includes probes for monitoring, but this is currently unimplemented.

Planit [6] leverages upon advances in the field of temporal planning. Given a model of the software system, a model of the network and a goal configuration, Planit uses a temporal planner to generate possible deployments of the system. Planit associates a duration with the reconfiguration actions, allowing its temporal planner to minimize factors such as the reconfiguration time. The goal configuration can include both specific predicates that describe a fixed desired state and non-specific predicates about the system that need to hold after the deployment finishes.

The ScudWare middleware platform [145] is a platform for smart vehicle spaces. It includes an adaptive component management framework. A framework for resource abstraction provides information about the lifetime, type and energy of components and devices. Based on this monitored information and based on quality-of-service requirements, the adaptive component management service migrates or replicates components.

The Jade [16] framework supports development of autonomic managers on top of legacy systems. The paper illustrates the framework by presenting an example for self-optimization. The autonomic manager in the example scales the number of servers used to handle the current load. The manageability of the legacy systems is achieved by wrapping the system in a component that enables configuration and reconfiguration.

Dashofy et al. [32] describe an architecture-based approach for self-healing systems. The approach is based on xADL 2.0, an XML-based architecture description language. The repair of a system uses repair diffs. In the overall vision of the paper, these repair diffs are generated by a fault detection and (re-)planning agent based on architectural constraints. However, the paper only discusses the infrastructure to reconfigure a running system by executing changes based on xADL and xADL diffs. The automatic planning of repairs is considered future work by the authors.

Georgiadis et al. [52] examine the feasibility of using architectural constraints as the basis for self-organizing architectures. The paper describes an execution environment for executing changes to the running system. This environment is completely decentralized, each component includes its own component manager that manages the architectural constraints for the component. Consistency of the architectural views in all the managers is maintained using atomic broadcast. The goal of the authors is to automatically derive the rules that govern the component managers from the declarative architectural constraints. However, the rules for the component managers in the paper were manually designed.

GEA and Adage [26] are two approximations of a generic model for the deployment

of grid applications. The generic model deploys applications based on a module description, deployment directives and additional information to guide the mapping of the application onto resources. This additional information consists of resource constraints, execution platform constraints, placement policies and a resource ranking. The resource ranking is a utility function provided by the user allowing the deployment tool to select the best mapping amongst the feasible alternatives.

This chapter presents a management solution for distributed applications in multi-purpose sensor networks. This tool, QARI, provides autonomous management of multiple applications on a wireless sensor network. QARI uses utility to guide its search for the best state of the sensor network and to decide on the trade-off between competing applications. QARI thus provides state-of-the-art features with respect to both the abstraction level and the degree of automation. To the best of our knowledge, QARI is the first to integrate such features in a tool for application management in multi-purpose sensor networks.

4.1.2 Utility in wireless sensor networks

Raising the abstraction level to the level of utility functions requires a way to asses the utility of a given state of the sensor network. This section provides some related work in that area. First, this section discusses MiLAN [60]. MiLAN confirms that multiple states of the network provide a sufficient utility level. The discussion continues with work in the area of coverage, a metric that is often used to express the utility of a wireless sensor network. Coverage is a metric that assesses how well the combined sensing ranges of the nodes in a sensor network cover the physical area that the sensor network is supposed to monitor.

MiLAN [60] addresses the topic of determining the optimal subset of sensors that is required to reach a desired quality-of-service level. MiLAN confirms that multiple subsets of sensors can provide the required data, but with different quality-of-service properties. The solutions in this chapter exploit the knowledge of the quality-of-service requirements to autonomously assign applications to a suitable subset of the sensor network.

The quality-of-service (surveillance) that can be provided by a particular WSN is commonly expressed in terms of k–coverage or related coverage metrics. Several papers in the literature discuss various aspects of this *coverage problem* in wireless sensor networks. Meguerdichian et al. [102] discuss several coverage related metrics and provide an algorithm to compute those metrics for a given sensor network. Based on their findings, they propose node deployment heuristics to achieve better coverage. Kumar et al. [86] discuss the problem of combining duty-cycling with a guaranteed k–coverage. They provide a theoretical analysis of three kinds of sensor network deployments: grid, random uniform and Poisson.

Wang et al. [140] present a *coverage configuration protocol* to configure the sleep scheduling of sensor nodes while maintaining different degrees of coverage requested by the application. Veltri et al. [138] present a localized algorithm that allows a sensor network to calculate its minimal exposure path (i.e. its worst-case coverage) and heuristics to calculate its maximal exposure path (i.e. its best-case coverage).

These approaches calculate coverage after the fact, propose node deployment heuristics based on theoretical coverage analysis or use coverage information for optimized sleep scheduling. This dissertation exploits the knowledge of (amongst others) coverage requirements to provide autonomous management of distributed applications in multi-purpose sensor networks.

4.2 Raising the abstraction level

This chapter aims to provide autonomous management of distributed applications on multi-purpose sensor networks through QARI. This is facilitated by raising the abstraction level of QARI's input policies in comparison to the abstraction level of the tools and application programming interfaces that runtime systems such as LooCI provide. The abstraction level is raised in two ways: (1) QARI aggregates components into applications and (2) QARI accepts deployment instructions at the utility level targeting groups of nodes, physical areas and the network.

The policies governing QARI's operation cover three types of information:

- Network information (Section 4.2.2): QARI needs information about the network it manages. This includes both static metadata (such as the type of the node) and initial values for dynamic properties (such as the node's initial location). It also includes information about the physical area in which the sensor network is deployed. Additionally, QARI accepts constraints on the network state that are independent of any application. This information is provided to QARI in the form of a **network description** and an optional **constraint specification**.
- Application information (Section 4.2.3): this type of information describes the assembly of applications from components and thus describes the application structure. A component-based description clearly describes the components in an application (a composition) and their dependencies (the wires). This information about the application's structure is provided to QARI in a **composition specification**.
- Deployment information (Section 4.2.4): the deployment information tells QARI how components should be assigned to elements in the network. It

thus connects the application information and the network information. This deployment information is at the utility level of abstraction allowing QARI to autonomously pursue the best assignment of components to nodes, even in the presence of network dynamics. The deployment information is provided to QARI in a **deployment specification**.

The stakeholders govern the behavior of QARI by supplying this information. Three types of stakeholders are identified that align with the three types of information above. Section 4.2.1 discusses these stakeholders and their concerns. The three sections thereafter discuss the three types of information in more detail and present the specifications describing that information.

4.2.1 Stakeholders

The rise in abstraction level of the policies benefits the autonomous capabilities of QARI. In addition, stakeholders can provide policies that align closer with the abstraction level of their mindset. To maximize convenience for the stakeholders, the policies are separated into multiple specifications in line with the principle of separation of concerns. Three stakeholder types are identified with different concerns in the application management in multi-purpose sensor networks: (1) application developers, (2) infrastructure administrators and (3) application administrators. Although three distinct roles are identified, a company or individual might embody more than one of these roles.

Application developers develop applications to run on sensor network infrastructure by composing existing and custom-made components. They create the software that QARI eventually deploys on a sensor network. Their interest in application management is limited to the description of the structure of the application. The specific sensor networks that will host their application doesn't concern them.

An infrastructure administrator is responsible for the management of (a part of) the sensor network infrastructure. Keeping the sensor network up and running is his major concern. The infrastructure administrator guards the availability and reliability of the network regardless of the applications on top. In order to prevent network overload, he solicits an influence on application management to constrain the behavior of individual applications.

An application administrator is responsible for the management of one or more applications. He determines the goals for the mapping of the application to the infrastructure. He decides on the utility functions that value the states of the network. However, the utility provided by the application administrator refers to application level concepts, rather than to network states. The details of the sensor network infrastructure are not a concern of the application administrator.

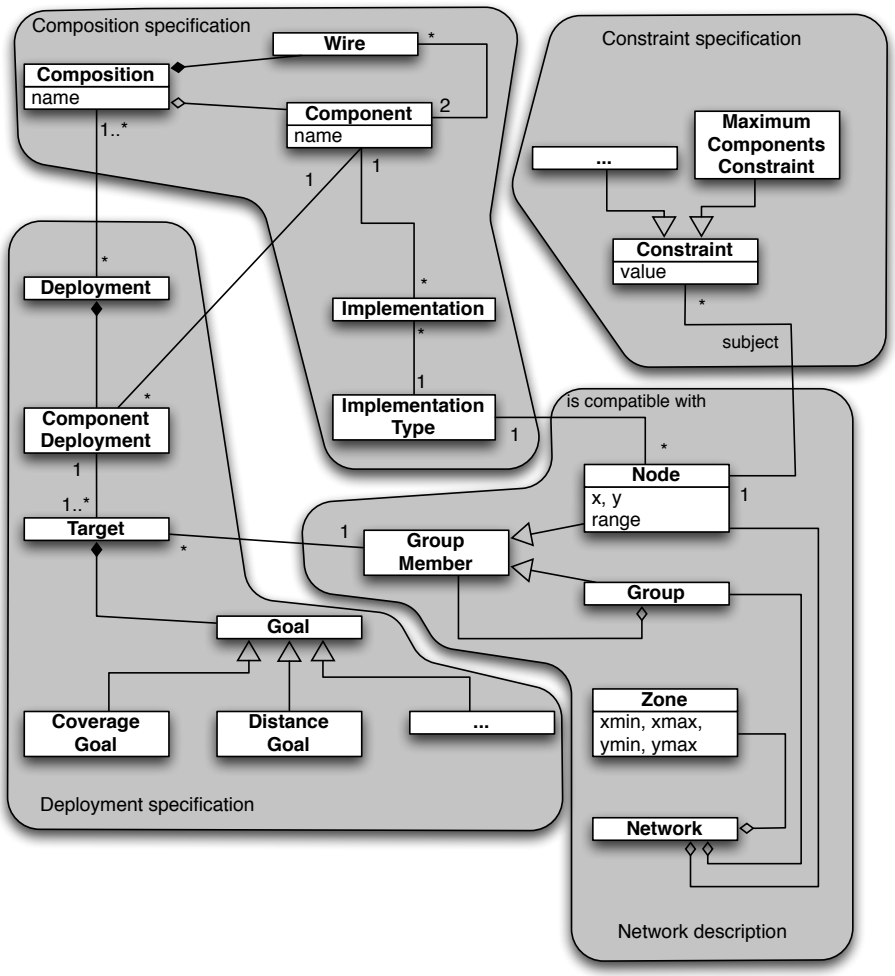


Figure 4.3: The domain model for the composition specification, constraint specification, deployment specification and network description.

The concerns of these stakeholders align nicely with the types of information that QARI requires. The network information in the *network description* and *constraint specification* is provided by the infrastructure administrators. The application developers deliver the application information in a *composition specification*. The application administrator expresses the goals for the deployment of an application in a *deployment specification*.

Figure 4.3 summarizes the concepts that are described in these four specifications. The following sections each detail one of the types of information. Section 4.2.2 describes the *network description* and the *constraint specification*. Section 4.2.3 zooms in on the *composition specification* and Section 4.2.4 details the *deployment specification*.

4.2.2 Network information

Infrastructure administrators supply information about the network they control. This information is twofold: (1) it contains information about the logical and physical structure of the sensor network, and (2) it details the constraints enforced by the infrastructure administrator. The former is the subject of the network description, the latter is contained in a constraint specification.

The network description contains information about the sensor network itself, as well as the physical location in which it is deployed. This information includes static metadata about the sensor nodes as well as initial values for their dynamic properties. The network description incorporates the following important concepts (see Figure 4.3):

Node First of all, the network description provides information about the nodes in the sensor network. This information includes the type of the node, the available sensors, the sensing range, etc. The description may also include initial values for dynamic properties, such as the location of the node.

Group A group contains group members, which can be either a node or another group. This allows hierarchical grouping of sensor nodes. For example, the infrastructure administrator can define a group of nodes with a temperature sensor available, a group of nodes with a humidity sensor available, and a group of nodes with either a temperature sensor or a humidity sensor available, i.e. a group that contains the two previous groups.

Zone A zone defines a geographical part of the network. Thus, a zone delimits a part of the physical area spanning the sensor network deployment. For example, when a sensor network is deployed on a floor of a building, zones could define the rooms on the floor. This information is needed to allow deployment goals that are specified in terms of a physical location (see Section 4.2.4).

Network The network aggregates the previous concepts. The network itself implicitly also defines a zone, i.e. the physical area spanning the whole sensor network. The description of the network therefore also contains information about the borders of this implicit zone.

The constraint specification defines the constraints that infrastructure administrators impose on the nodes under their responsibility. The state of the network must satisfy these constraints to be compliant with the infrastructure administrators policies. An administrator can, for example, put a limit on the number of components on a single node. The constraints give the infrastructure administrator control over certain non-functional properties of the network, such as availability, reliability and security. The network administrator is responsible for managing these non-functional properties, regardless of the applications running on the sensor network.

A constraint consists of a type, a subject and a value. The subject of the constraint is a sensor node. The interpretation of the value depends on the type of the constraint. The type corresponds to an expression over the node's properties yielding a value that is to be compared with the value in the constraint. For example, the maximum components constraint shown in Figure 4.3 expresses the constraint: the maximum number of components simultaneously active on *subject* should be less than or equal to *value*.

Various types of constraints can be useful, but the constraints should be checkable by QARI. Therefore the expression over the node's properties should be computable based on information that is available to QARI. This information must thus be (1) a static property provided in a policy, (2) monitored by QARI or (3) generated by QARI itself. The maximum components constraint above is computable by QARI because the information about which components are active on each node is generated by QARI. Examples of constraints that involve monitored information are: only activate new components on *subject* as long as the memory usage or network load is less than *value* and as long as the battery level is more than *value*.

4.2.3 Application information

The application developer describes the structure of an application in a composition specification. This specification describes the components that constitute an application and details the way these components are composed into an application. This includes information about the wires, i.e. the connections between the components. This specification thus describes the following concepts (see Figure 4.3):

Composition A composition is a collection of components that is connected through wires. A composition has a unique name that identifies the composition. A composition represents an application of cooperating components that collectively realize the applications functional and non-functional requirements.

Component The component is the basic unit of encapsulation in the component specification. Similar to a composition, a component is uniquely identified by its name. A component has at least one implementation that realizes the component.

Wire A wire is an artifact that connects two components. A wire only exists in the context of a composition. A wire always connects two components, but a component can be wired multiple times.

Implementation An implementation is the realization of a component. It is binary code that can be deployed to a node. A component can have multiple implementations of different types. The implementation type implicitly defines the runtime support needed to run the implementation. A node is compatible with an implementation type if the node provides this runtime support (expressed by the node type). A component is deployable to a node if it has at least one implementation that is compatible with the node.

These concepts describe the basic information that QARI needs about the application in order to successfully plan a deployment of the application. Additional information is often needed to execute that deployment. For example, multiple mechanisms might be available to wire two components. The information needed to determine the preferred mechanism should also be included in the composition specification. The Service Component Architecture (SCA) Assembly specification format [110] is a vendor-neutral format to describe compositions that includes support for the basic concepts discussed above. The SCA assembly specification format also provides the necessary features to describe the additional information. The prototype implementation in Chapter 5 therefore uses the SCA assembly specification format for the composition specification.

4.2.4 Deployment information

The application administrator produces a deployment specification that defines the targets for the components of an application. This is the specification that maps the application components in the composition specification to the nodes and groups in the network description. The specification includes quality goals for each mapping. From these goals QARI derives a utility for each possible state of the network. The utility then guides the autonomous behavior of QARI. The concepts in the deployment specification (see also Figure 4.3) are:

Deployment A deployment is the container for the deployment related information of one or more compositions. It aggregates component deployments.

Component Deployment A component deployment maps a component to one or more targets. A component deployment is part of a deployment.

Target The target of a component deployment consist of (1) a node or group of nodes and (2) an optional set of goals. The set of goals is optional because a goal is useless if the target is a single node. In that case, planning is not needed, thus goals to govern that planning are superfluous.

Goal A goal describes the required quality levels that the planner should satisfy. A goal contains both a nominal quality level and a minimal quality level. These quality levels define a utility function that scores each possible outcome of the planning. Outcomes that provide a quality level above the nominal level are preferred. Outcomes with a quality level below the minimal level are the least desirable. A quality level between the minimum and nominal level is acceptable if the nominal value is not reachable. For specific types of goal, more fine-grained utility functions can be defined.

Goals define limits for properties relevant to the application. Consider, for example, fraction goals, coverage goals and distance goals:

- Fraction goals require the size of the set of selected nodes to be above a threshold percentage of the total size of the target group (e.g. deploy to 50% of the nodes).
- Coverage goals require the combined sensing range of the selected nodes to cover a given area of the network a given amount of times (e.g. deploy to a subset of the nodes, such that zone X is covered twice).
- Distance goals limit the distance between the selected nodes and the nodes hosting another component (e.g. deploy to nodes less than 2 meter away from nodes that run component Y).

Each type of goal requires support in the planning component of QARI for calculating the associated utility.

These quality goals relate to the data quality that the applications must support. Data quality has two important aspects: temporal quality and spatial quality. Only the spatial quality is important in the context of the management solution in this chapter. Based on the spatial quality requirements expressed in the goals, QARI autonomously decides which subset of the targeted nodes will host a component. QARI's decisions do not influence the temporal quality, which explains the absence of temporal quality goals in QARI's deployment specifications.

4.3 Multi-application support

The goal of this chapter is to offer autonomous management of multiple applications on multi-purpose sensor networks. Section 4.2 illustrated how the abstraction level is raised to facilitate autonomy. This section discusses how QARI manages multiple concurrent applications.

Supporting multiple applications implies the need to handle potentially competing applications. Whenever two or more applications compete for the same sensor network resources, a trade-off must be made. Also, multi-application support means taking into account the current state of the sensor network when a new request for managing an application arrives. One cannot assume a clean slate network when planning the deployment of an application.

QARI's multi-application support starts with accepting multiple independent policies each describing one or more applications. Application managers can independently submit their deployment specifications. QARI takes care of deriving a single target state for the sensor network from these specifications. QARI strives to optimize the overall utility of this target state for the complete sensor network.

The overall utility combines the utilities of the individual applications in a single metric. This is a big advantage of utility-based policies over the other levels of abstraction: utility-based policies are composable. The other abstraction levels do not possess this feature. For example, the criteria of two independent applications may describe two non-overlapping sets of desired states. In such case, it is not clear from which set the final target state should be drawn.

Multi-application support blurs the difference between planning and re-planning. Except for the first deployment request, the planning of the deployment always has to take previously deployed applications into account. This is similar to the re-planning after, for example, a node failure which also has to take into account all deployed applications. Since performing changes to a running network can interfere with the operation of the running applications, it is important for re-planning to find a new target state that balances good utility and minimal changes to the network. Therefore, QARI enriches the overall utility with a system-level utility that favors network states with minimal changes to the current state.

Taking the other applications into account when planning the deployment of a new application also provides benefits. Because QARI knows what components the applications need where, it can share the same component between two or more applications. On the other hand, when sharing is not possible, QARI evenly spreads the components of the applications to balance the load in the sensor network.

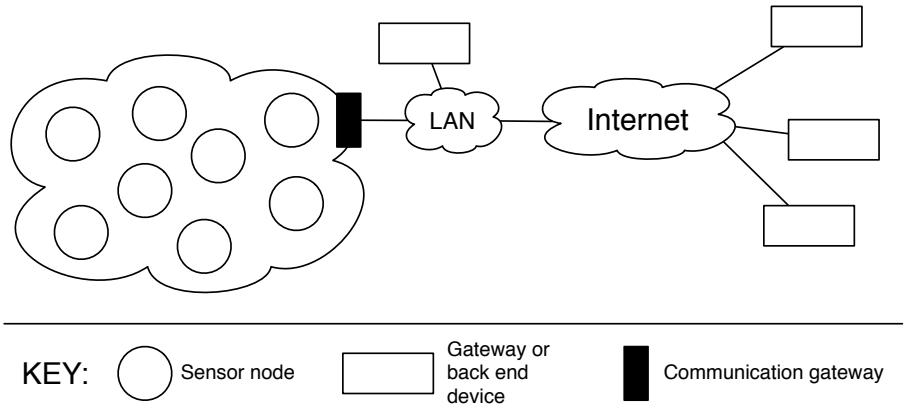


Figure 4.4: Model of the infrastructural context in which QARI operates.

4.4 Architecture

This section describes the architecture of the autonomous management tool for distributed applications on multi-purpose sensor networks (QARI). Section 4.4.1 first describes the context and system boundaries of QARI. Then, Section 4.4.2 provides a high-level overview of the architecture. Finally, Section 4.4.3 provides detailed descriptions of the components and interfaces in QARI's architecture.

4.4.1 Context

This section describes the context and defines the system boundaries of QARI. QARI operates in the infrastructural context shown in Figure 4.4. A multi-purpose sensor network is connected to the Internet via local infrastructure. Several back-end systems interact with the sensor network via the Internet.

Depending on the situation, the local infrastructure differs greatly. Some use cases require only a single gateway, other scenarios demand a local area network with plentiful computational power, though most scenarios need something in between these extremes. When QARI is used, the local infrastructure should provide enough computational power to support QARI's needs (see Section 5.4.3).

Figure 4.5 shows the system boundaries of QARI. On the one hand, QARI interacts with other management systems or human operators, typically located at back-end systems. The stakeholders supply the specifications discussed in Section 4.2 via

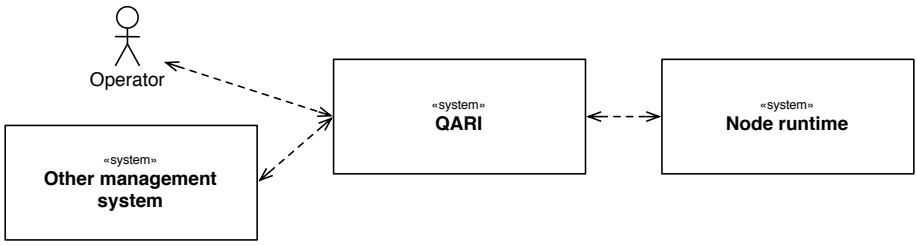


Figure 4.5: System boundaries for QARI.

these interactions. On the other hand, QARI interacts with the runtime system on the sensor nodes for the execution of management actions in the sensor network.

To conclude, QARI operates on a node in the local infrastructure at the edge of the multi-purpose sensor network. QARI interacts with:

- other management systems (or human operators).
- the node runtime systems (e.g. LooCI).

4.4.2 Architecture overview

Figure 4.6 shows an overview of QARI's high-level architecture. The high-level operation of QARI is inspired by the MAPE-K reference architecture for autonomic computing [71]. This reference architecture describes methods to structure the functionality needed for autonomous management of a target system. In this case, QARI manages distributed applications on a multi-purpose sensor network. The nodes in the network should host a reconfigurable, component-based runtime system, such as LooCI (see Chapter 3). The main building blocks of QARI are the *Information Repository*, the *Monitoring* component, the *Analyze* component, the *Planning* component and the *Execution* component.

The *Information Repository* component centralizes all knowledge in QARI. It aggregates the specifications (see Section 4.2) and all information collected by the other components. The *Information Repository* makes this information available for querying by the other components. This includes information about the goals of the administrators, information about the target state of the sensor network (as resolved by the *Planning* component) and information about the current state of the sensor network. In a stable state, the actual state of the sensor network matches the target state computed by QARI.

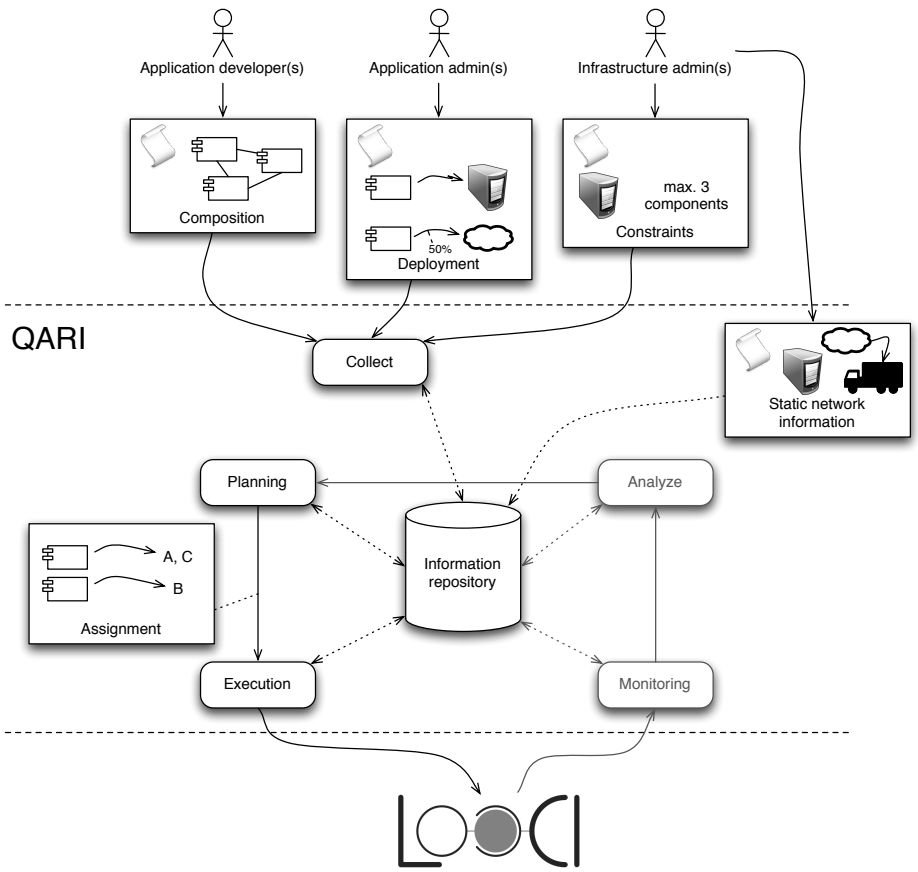


Figure 4.6: Overview of QARI

The *Monitoring* component monitors the sensor network for potentially interesting information and events that enable QARI to assess the current state of the sensor network. For example, the *Monitoring* component might collect information about the availability of a node through polling, through detecting a heartbeat or through detecting data traffic originating at that node.

The *Analyze* component analyzes the information from the *Monitoring* component and extracts meaningful events from the stream of monitored information. It detects, for example, node failures by correlating different information streams related to the availability of a node. It can combine historic events with theoretical models to predict node failures. Analysis can also discriminate between a node failure and a node disconnection.

The task of the *Planning* component is to plan changes to the managed system whenever new policy specifications are submitted or whenever a disrupting event is detected by the *Analyze* component. QARI's *Planning* component calculates assignments based on the specifications (see Section 4.5.1). An assignment maps an application component to a set of nodes that should host an instance of that component. The deployment specifications contain instructions that target application components to a group of nodes or even the whole network. Such target includes goal information to govern QARI's *Planning* component. This goal information enables QARI to calculate the utility of an assignment.

The *Execution* component executes the necessary changes to the managed system, i.e. it executes management actions on the sensor network through the support of the nodes' runtime system. Through the execution of these actions, the *Execution* component evolves the sensor towards the state expressed in the assignments. Therefore, the *Execution* component deploys, removes, connects and disconnects components until the components on the sensor nodes match the assignments computed by the *Planning* component.

4.4.3 Architecture details

Figure 4.7 shows the architecture of QARI in more detail. The remainder of this chapter focuses on planning and execution, including the information repository that provides information storage to these components. Figure 4.7 shows the components in the *QARI Planning and Execution* subsystem, along with the *Information Repository* component.

Information Repository

The *Information Repository* is a repository that stores all information relevant for any of the other components. It provides a CRUD interface (create, retrieve,

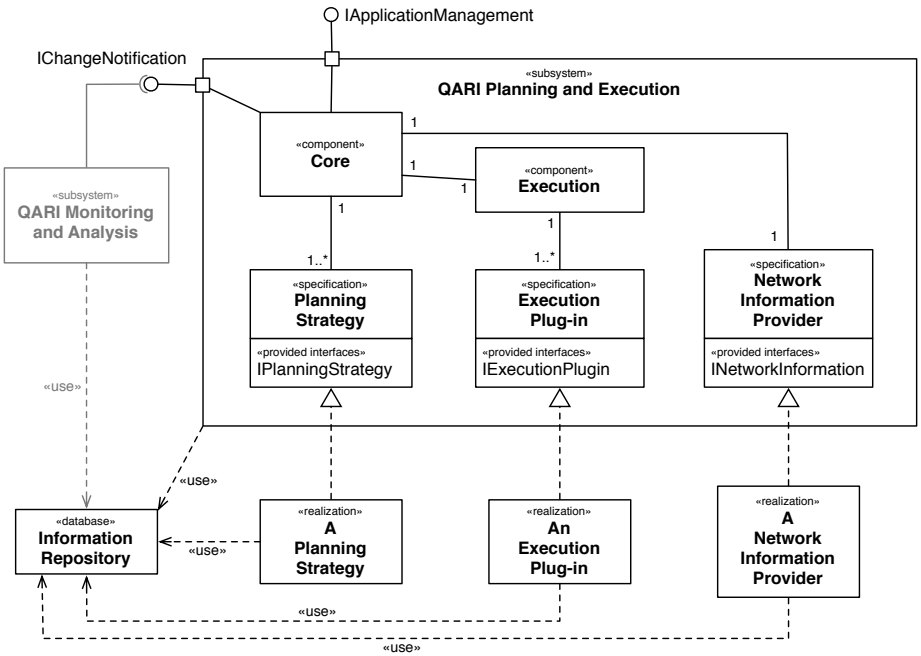


Figure 4.7: Detailed view of QARI's architecture.

update, delete) for this information. All information sharing in QARI happens through the *Information Repository*. The *Information Repository* manages three types of information:

- Policy information: the first type of information included in the *Information Repository* is the information in the input specifications expressing the desires and goals of the end-users. This includes the information in the composition, deployment and constraint specifications.
- Target network state: the result of the planning is a set of assignments that expresses the target state of the sensor network. This target state is also stored in the *Information Repository*.
- Current network state: the repository also contains information on the current state of the sensor network. This includes static network information (node metadata, physical layout, etc.), monitored information (node status, node location, etc.) and information about the results of the *Execution* component.

```
1 void handleNodeFailure(NodeID node);
2 void handleNodeAppearance(NodeID node);
3 void handleNodeMovement(NodeID node, Location newLoc);
```

Listing 4.1: Pseudocode for interface: IChangeNotification.

In a steady state, these three types of information are consistent with each other. The target state is the result of the planning phase and satisfies all desires and goals included in the policy information. The execution phase updates the network until the current network state matches the target state.

Planning and Execution

The planning and execution subsystem is built as flexible as possible to allow for adaptation to the specific context in which QARI is used. It consists of (1) a *Core* that coordinates the other components, (2) *Planning Strategy* plug-ins, (3) an *Execution* component and *Execution Plug-in* components and (4) a *Network Information Provider*.

Core. This component controls the planning and execution process by managing the operation of the *Planning Strategy* plug-ins and the *Execution* component. It provides two interfaces:

- IChangeNotification (Listing 4.1): This interface allows the *QARI Monitoring and Analysis* subsystem to signal interesting events. These events indicate a change that may require re-planning. Listing 4.1 shows three examples of such events: node failure, node (re)appearance and node movement.
- IApplicationManagement (Listing 4.2): This interface enables management of the policies that govern planning. Human operators or higher-level management systems use this interface to interact with QARI. The interface allows to add, update or remove deployment and composition specifications. The *Core* updates the specifications in the *Information Repository* accordingly and initiates planning when necessary.

Planning Strategy. *Planning Strategy* plug-ins perform the majority of QARI's planning. The rationale for making this a plug-in is the observation that *the optimal planning strategy* is non-existent. The best strategy for planning

```
1 void addComposition(CompositionSpecification newComp);
2 void addDeployment(DeploymentSpecification newDepl);
3 void removeComposition(QName compID);
4 void removeDeployment(QName deplID);
5 void updateComposition(CompositionSpecification updatedComp);
6 void updateDeployment(DeploymentSpecification updatedDepl);
```

Listing 4.2: Pseudocode for interface: IApplicationManagement.

```
1 Set<Assignment> updateAssignments(Set<Assignment> initial);
```

Listing 4.3: Pseudocode for interface: IPlanningStrategy.

a deployment depends on the specific context in which QARI is used (see Section 4.5.1). A plug-in for planning must implement the following interface:

- IPlanningStrategy (Listing 4.3): The single method in this interface instructs the plug-in to update a given set of initial assignments to a new set of assignments. The planning strategy generates the new set of assignments so that it is consistent with the policies. The plug-in retrieves any other information it needs from the *Information Repository*.

The *Core* passes an initial set of assignments to the plug-in for two reasons. First, as highlighted in Section 4.3, planning will almost never target a clean slate network because of the multi-application support. The initial set serves as the starting point for the planning. Second, the interface is designed that way to allow chaining of the plug-ins. Chaining the plug-ins can be done by passing the output of one plug-in as the initial set for a second plug-in. This is useful because some planning strategies perform better (i.e. they produce a solution with a higher utility and/or produce a solution faster) when a good initial solution is available. Using chaining, this initial solution can be produced by a less accurate, but faster planning strategy.

Execution and Execution Plug-in. The *Execution* component takes the necessary actions to ensure the actual state of the sensor network matches the target state produced by the planning. The *Execution* component produces the list of necessary actions by comparing the current state and the target state as stored in the *Information Repository*. By comparing the set of components that is assigned to a particular node with the set of components that is currently deployed on the node, the *Execution* plug-in produces a list of *deploy component* and *remove component* actions. In addition, the wires that have one of these components as source or target will have to be created or removed respectively. The resulting

```
1 Set<ComponentInstance> deploy(Component component, Set<Node>
      nodes);
2 boolean remove(ComponentInstance componentInstance);
3 Set<WireInstance> wire(Component source, Component target,
      Set<Node> sourceNodes, Set<Node> targetNodes);
4 boolean unwire(WireInstance wireInstance);
5 boolean canDeploy(Component component);
6 boolean canRemove(ComponentInstance componentInstance);
7 boolean canWire(Component source, Component target);
8 boolean canUnwire(WireInstance wireInstance);
```

Listing 4.4: Pseudocode for interface: IExecutionPlugin.

list of actions is only partially ordered: wiring a component has to be done after it has been deployed and removing a component has to be done after it has been unwired. The *Execution* component can order the execution of the actions as it sees fit as long as these two constraints are satisfied. There are several criteria that may be used to order the execution of the actions (see Section 4.5.2).

To address potential heterogeneity of the managed sensor network, the *Execution* component uses one or more *Execution Plug-in* components to interact with the nodes' runtime systems. The interface of such plug-in is:

- IExecutionPlugin (Listing 4.4): This interface allows the *Execution* component to execute software management actions on nodes supported by the given plug-in. The interface provides methods to deploy, remove, wire and unwire components (lines 1–4). In addition, methods to check the capabilities of a given plug-in are available (lines 5–8). The methods to deploy and wire components return information about the deployed components or the created wires. This information is subsequently stored in the *Information Repository* where it is available for future planning. This information is also needed as the input for the remove and unwire methods. The wire method takes two sets of nodes as input: each source node will be wired to each target node, resulting in a number of wires equal to the number of source nodes times the number of target nodes.

Network Information Provider. The *Network Information Provider* abstracts the details of network information retrieval. This information contains the static network information in the network specifications and the constraints in the constraint specification. The network administrator provides these specifications in which he describes the physical layout of the network, metadata about the nodes and constraints that apply to the nodes. An instance of QARI that manages a

```
1 void retrieveNetworkSpecification(Set<String> networkNames);
2 void retrieveConstraintSpecification(Set<String>
      networkNames);
```

Listing 4.5: Pseudocode for interface: INetworkInformation.

closed network (i.e. a network where nodes can fail, reappear and move, but where no new nodes enter the network), can read this network information from a configuration file. In the case of a QARI instance that manages a network where nodes leave and join the network, this information may be fetched on demand from an information source on the Internet. To shield the *Core* from these variations, the architecture uses the principle of *protected variations* and therefore isolates this behavior in a plug-in. The interface for this plug-in is:

- INetworkInformation (Listing 4.5): Using the two methods in this interface, the *Core* instructs the *Network Information Provider* to retrieve network information and constraints for a given set of network elements and store it in the *Information Repository*. The network elements are identified via a network name. This network name can be the name of a network, a node identifier or a group identifier.

The *Network Information Provider* makes the information available to the other components by storing it in the *Information Repository*. After the initial retrieval of this information, the *Network Information Provider* may keep it up-to-date by periodically polling its source for updates or through any other mechanism available. The *Network Information Provider* shields these details from the other components, the latest network and constraint information is always available to them in the *Information Repository*.

4.5 Discussion

This section discusses the planning and execution of a deployment on multi-purpose sensor networks. First, it defines the planning problem that QARI must solve. The discussion then continues on the two planning strategies explored in this research. Then, this section describes how planning can take wireless sensor network specifics into consideration. Finally, the degrees of freedom in scheduling deployment actions are discussed. The execution component can exploit this freedom to realize certain objectives, such as limiting the burden on the network.

4.5.1 Planning deployment

The planning problem. The problem QARI has to solve during planning is: find an assignment of components to nodes so that the constraints in the constraint specifications are satisfied and the utility of the solution is maximized. Assume C is the set of components, N is the set of nodes and A is the set of assignments for which the constraints are satisfied, then:

$$A \subseteq 2^{C \times N}$$

Where:

- $C \times N = \{(c, n) : c \in C, n \in N\}$
- $2^{C \times N}$ is the powerset of $C \times N$, i.e. the set of all subsets of $C \times N$

The goals in the deployment specifications and the system-level objectives (such as minimize the number of nodes needed) determine the desirability of a solution. They can be expressed as utility functions U_i that assign a utility value to each possible solution:

$$U_i : A \to \mathbb{Z}$$

If S is the solution QARI is searching for, then the problem to solve is:

$$\max_{S \in A}(U_i(S), \cdots, U_n(S))$$

This problem is a multi-objective combinatorial optimization problem. The survey in [44] gives an overview of methods to solve such problems. Some methods are exact methods: these methods solve the problem as stated above. Because the problem in its general form is NP-complete, approximation methods have been developed which seek a good solution within acceptable computational bounds. These methods, often called heuristics and meta-heuristics, do not guarantee feasibility or optimality.

It is not in the scope of this dissertation to improve upon the state-of-the-art in methods for solving multi-objective combinatorial optimization. The work in this dissertation is limited to the use of existing methods and frameworks for addressing the planning needs.

```
(1)  assignment := empty set
(2)  areas := all areas in zone X
(3)  sort areas by number of covering nodes
(4)  for area ∈ areas do
(5)      candidates := nodes covering area
(6)      sort candidates by score
(7)      while area.coverage < required coverage ∧ candidates not empty do
(8)          candidate := candidates.remove(0)
(9)          if candidate ∉ assignment
(10)           then assignment.append(candidate)
(11)         fi
(12)     od
(13) od
```

Figure 4.8: Pseudocode for a heuristic algorithm to solve the simplified planning problem.

Planning strategies. It is not immediately clear what method performs *best* to solve the planning problem in QARI. It heavily depends on the specific context in which QARI is used. What types of goals are expected? What property receives more importance: planning speed or optimality of the result? To address this uncertainty, QARI's architecture includes a plug-in system for planning strategies (see Section 4.4.3). The following paragraphs discuss two planning strategies explored in this research.

The first strategy applies a problem specific heuristic to the design of an algorithm that constructs a good solution. For simple variants of the planning problem, designing such algorithm is straightforward. Consider the assignment of a single component to the nodes in a network. There is one coverage goal that requires zone X to be covered. As a system-level objective, the algorithm tries to assign the least amount of nodes needed. To calculate the coverage in this example, zone X is divided in a set of areas of fixed size. Zone X is considered k-covered if for each area, there are at least k nodes that cover that area. A node covers an area if the sensing range of the node contains the area.

Figure 4.8 shows an algorithm that constructs a good solution for this problem. Note that this algorithm constructs a *good* solution, but not necessarily the *best* solution. The algorithm orders the areas in zone X based on the number of nodes that cover the area. It then considers each of the areas in this order. For each area, the algorithm adds the *best* nodes to the assignment until the area is sufficiently covered. The *best* nodes are the nodes that (1) are not already in the assignment and (2) have the best score. The score is calculated based on (1) the number of (not yet covered) areas the node covers, (2) the number of components already

running on the node and (3) the reputation of the node. The reputation of a node is used to penalize nodes that have failed in the recent past.

This algorithm is simple and easy to understand, but it is specific to this simplified planning problem. Extending the algorithm to support other types of goals or system-level objectives is non-trivial. Meta-heuristics provide more generally applicable techniques and are thus easier to extend with support for multiple types of goals.

Therefore, the second planning strategy leverages a framework that implements methods based on meta-heuristics [78]. Most methods require a reformulation of the problem as a single objective combinatorial optimization problem [44]. This reformulation replaces the utility functions by a single utility function that is a weighted sum of the individual utilities. The planning problem thus becomes:

$$\max_{S \in A} \sum_{i=0}^{n} w_i U_i(S)$$

A local search algorithm (tabu search [115]) provided by the framework is applied to solve QARI's planning problem. Local search starts from an initial solution and tries to improve upon that solution by following a single path through the solution space. At each iteration, it evaluates a number of moves from the current solution to a potential next solution. It then selects the best move and applies it to the current solution. It repeats this until it reaches an optimal solution (if the optimal value of the utility function is known in advance) or until a stop criterion is satisfied. Possible stop criteria include: there have been a certain number of iterations without an improvement in the solution, a predefined total number of iterations is performed or a configured time limit is exceeded.

WSN specifics. Planning strategies can support system-level objectives during planning. Therefore they incorporate utility functions that represent those system-level objectives, in addition to the utility functions derived from the goals in the deployment specifications. In the context of multi-purpose sensor networks, several system-level objectives can improve the overall lifetime or quality-of-service of the network. For example, to extend the lifetime of the sensor network it is beneficial to meet the user goals with the least amount of nodes. Evenly distributing the components over all nodes can also help in this context because it avoids overloading a single node. Another system-level objective might favor allocation of components to nodes with more resources. Because communication from nodes far away from the gateway uses more energy for forwarding, the network lifetime also benefits from assigning to nodes close to the gateway.

The goals in the deployment specification are provided by the application administrators of the respective applications. In multi-purpose sensor networks,

multiple applications from different application administrators have to be deployed simultaneously. Because the goals for the applications originate from independent application administrators, the applications might compete for the same resources when brought together for deployment on the same sensor network. In that case, it might be impossible to satisfy the goals of all applications. The planning will then come up with a solution that maximizes the overall utility. To prioritize certain applications, an infrastructure administrator assigns more weight to the utility functions that correspond to the goals of those applications.

After a node failure or another event that requires re-planning, it is better to minimize the number of new deployment actions. Deploying a component to a node implies transmitting the code for that component to the node. Since radio communication is a costly operation in terms of energy use, deploying a component is also costly. The planning strategy should thus try to minimize the changes to the network when re-planning. A planning strategy can achieve this by (1) using the current assignment as the initial solution and (2) adding a system-level objective that discourages new component deployments.

The effects of the integration of these WSN specific objectives into the planning can be further improved when richer cost metrics are available. For example, if the energy cost for deploying a component in function of the distance to the gateway is known, a planning strategy can be designed that takes this metric into account to decide whether it is better to (1) deploy two components close to the gateway, or (2) deploy one component further away from the gateway.

4.5.2 Executing deployment

The result of the planning in QARI is at the abstraction level of a single desired state and thus declarative. As discussed in Section 4.4.3, the execution component derives the necessary action by comparing this desired state with the current state of the network. This results in a partially ordered list of actions. The order of the actions does not influence the result as long as two conditions are satisfied: (1) a component has to be deployed before it can be wired and (2) a component has to be unwired before it can be removed. As a consequence, the execution component can schedule the actions freely as long as these two conditions are met. It can execute some actions in parallel or it can use sequential execution of the actions. The execution component can exploit this scheduling freedom to achieve certain objectives. For example:

- Phased deployment: when a component must be deployed to a large group of nodes, the execution component deploys the component sequentially to smaller subgroups. Even if the runtime system deploys new components without direct impact on existing components, doing so will have certain

side effects. The component must be transmitted to the node, which will cause increased network traffic. The node also needs processing resources to install the component upon arrival. Phased deployment limits the overhead and disturbance on the network.

- Minimal quality level first: the execution component prioritizes certain deployments to reach the minimal quality level for a component as soon as possible. The component is first deployed to a subset of the assigned nodes that results in the satisfaction of the minimal quality level. After the minimal level is reached, deployment to the rest of the nodes in the assignment starts.

When several of these objectives need to be accounted for, scheduling the actions becomes a complex problem. Techniques from the domain of AI planning, such as the ones used by El Maghraoui et al. [47] or Pegasus [15, 35], can be used to perform this scheduling.

4.6 Summary

This chapter presented a management solution for assembly and deployment of distributed applications in multi-purpose sensor networks. This solution leverages the component-based runtime system in Chapter 3 (LooCI). The solution improves the manageability by enabling administrators to express the goals for their applications using a high-level abstraction. It provides a management tool, QARI, that autonomously pursues these goals. QARI's first contribution, application assembly, is an essential enabler for its main contribution: autonomous deployment of multiple applications. QARI automatically generates deployment patterns that satisfy the goals of the application administrators. After deployment, QARI monitors the network and updates the deployment in response to disrupting events.

Section 4.1 first identified four levels of abstraction in the input policies for automated software deployment tools: (1) action policies that describe what to do, (2) policies that describe a desired state, (3) policies that describe criteria for the system state and (4) policies that define a utility function to rank system states. The management tool in this chapter expects policies at the utility policy level because this eases the trade-off between multiple goals. The tool achieves such trade-off by optimizing the overall utility of the target state.

Next, Section 4.4 introduced the architecture of the proposed autonomous management tool (QARI). The architecture is inspired by the MAPE-K reference architecture. This dissertation then focused on the planning and execution subsystem of the architecture and provided detailed descriptions of the components

and interfaces in this subsystem. For planning, the architecture includes a plug-in system for planning strategies that enables tailoring of the planning to a specific context. The execution plug-ins enable support for a variety of runtime systems.

The planning problem that QARI must solve is an instance of a multi-objective combinatorial optimization problem. Several methods to solve this class of problems exist and this thesis explored two options: a deterministic algorithm using a problem specific heuristic and a meta-heuristics based local search algorithm. The discussion then continued about how sensor network specific issues can be taken into account in QARI's planning strategy. Finally, the text described how the execution component can reorder the deployment actions to achieve system-level objectives, such as limiting the impact of a deployment on the overall sensor network.

Chapter 5

Prototype, illustration and evaluation

This chapter presents prototype implementations of (1) the runtime support for reconfigurability in Chapter 3 and (2) the management solution for assembly and deployment of distributed applications in Chapter 4. This chapter discusses the implementation of the DAViM and LooCI runtime systems on state-of-the-art sensor network hardware and the implementation of QARI on a gateway platform (See Figure 5.1).

The features of these prototypes are illustrated in scenarios in the context of transport and logistics. Two representative examples of the use of multi-purpose sensor networks in transport and logistics are discussed. The first scenario considers warehouse monitoring using a sensor network. The second scenario applies sensor networks to the monitoring of a truck and trailer combination.

This chapter also reports on the integration of the prototypes of QARI and LooCI in a testbed with state-of-the-art sensor nodes. The prototypes and their integration in a physical testbed confirm the feasibility of this dissertation's approach. The measurements collected from these prototypes in simulation and on the testbed show the resource efficiency of the presented management solutions for distributed applications on multi-purpose sensor networks.

This chapter starts with the introduction of the two example scenarios in Section 5.1. Section 5.2, Section 5.3 and Section 5.4 discuss and illustrate the prototype implementation of DAViM, LooCI and QARI respectively. Finally, Section 5.5 reports on the integration of the prototypes of QARI and LooCI.

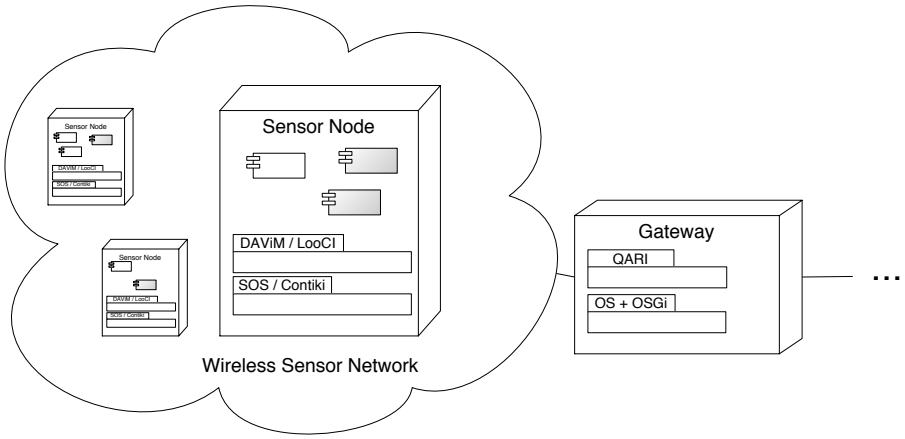


Figure 5.1: Positioning of the prototypes of DAViM, LooCI and QARI.

5.1 Example scenarios

This section introduces some example scenarios in the context of transport and logistics that are used to illustrate the prototypes in this chapter. First, this section discusses the application context of transport and logistics (Section 5.1.1). It then describes two example scenarios: warehouse monitoring and truck/trailer monitoring (Section 5.1.2). These scenarios are inspired by the scenarios used in several research projects with industrial partners in the transport and logistics sector [69, 125, 70]. Section 5.1.3 concludes with an illustration of the input policies for QARI in the truck/trailer monitoring scenario.

5.1.1 Application context: transport and logistics

The transport and logistics domain involves a broad range of stakeholders. A transport company carries goods for companies and individuals along the complete supply chain. These companies and individuals include suppliers, manufacturers, wholesalers, retailers and consumers. These parties contract the transport company to deliver timely and good quality transport of their goods. Good quality transport implies correct handling of the goods, including appropriate environmental conditions.

Apart from these direct stakeholders, the transport and logistics industry also involves external parties. The government, for example, influences the sector

through taxes on fuel, customs regulations, environmental regulations and road tolling. Harbor and airport operators govern the handling of cargo in their respective facilities. Additionally, a transport company might call in contractors in the process. For example, assuring the physical security of the premises is typically outsourced to a specialized company.

Emerging regulatory and market pressure forces actors in the transport and logistics industry to invest in (technological) solutions that improve the visibility and quality control throughout the supply chain. For example, the container security initiative (CSI) [136] of the U.S. federal government, proposes the use of smarter, more secure containers to improve border security. The status of Authorized Economic Operator (AEO) [48] in the European Union provides access to simplified customs procedures. The criteria for the status of AEO include a satisfactory system of managing commercial and transport records and compliance with security and safety standards.

Wireless sensor network technology is an eligible candidate to meet the requirements of tracking and tracing, status monitoring and physical security surveillance in a transport and logistics context. These applications require sensing capabilities such as location tracking, temperature measurements, humidity monitoring, vibration detection, light monitoring and door sensors.

The domain of transport and logistics is thus a representative application context for multi-purpose sensor networks. The sensor networks in transport and logistics are a reusable asset hosting multiple applications that evolve due to the changing regulatory and market pressure and due to the varying types of cargo. The transport and logistics domain implies a multi-party environment where each stakeholder has potentially different monitoring requirements.

5.1.2 Example application scenarios

This section presents two representative example scenarios in the transport and logistics context (see Section 5.1.1). It first introduces a warehouse monitoring application and then continues with a truck/trailer monitoring scenario. These two scenarios cover the two basic activities in the transport and logistics context: storing goods and transporting goods. Of course, several other storage and transport options exist, but this section is limited to one example in each category. Other examples include outdoor storage on a harbor dock and transportation over sea or by train.

Warehouse monitoring. This scenario deals with the application of sensor network technology to monitor a warehouse. This includes monitoring of both the goods stored inside the warehouse and the warehouse itself. The sensor network

in the warehouse consists of fixed sensor nodes and sensor nodes attached to pallets and containers inside the warehouse. Potential applications for such sensor network include:

- Tracking and tracing of goods: tracking and tracing concerns discovering and logging the location of a container, pallet or box. For items that have sensor nodes, node localization techniques [88] can be employed. This is especially interesting in indoor situations, such as inside warehouses, where GPS positioning is not available.
- Monitoring of safety perimeters: if a warehouse stores hazardous products, safety perimeters are vital for the prevention of accidents. The sensor network can monitor the distance between products and raise alarms when appropriate.
- HVAC: heating, ventilation and air conditioning are essential features of every facility management solution. To operate properly, an HVAC system needs accurate information about the environmental conditions inside the warehouse. The wireless sensor network can complement the dedicated HVAC sensors to improve the accuracy of the monitoring.

To illustrate the use of this dissertation's management solutions in this scenario, consider a temperature monitoring application that supplies detailed temperature information to the HVAC system. For accurate measurements, the temperature sampling component must be deployed to a set of nodes that covers the warehouse area. Sensor nodes attached to pallets and containers move throughout the warehouse whenever their carrier is moved. These nodes also join and leave the network since the warehouse only stores pallets and containers temporarily. This dynamic nature of the sensor network requires continuous re-evaluation of the coverage. The need for continuous monitoring and reconfiguration renders manual management infeasible. QARI automates this planning and re-planning under dynamic network conditions and the modular runtime systems (DAViM, LooCI) allow to reconfigure the temperature monitoring application with minimal disruption to the other applications.

Truck/trailer monitoring. The second example scenario applies sensor network technology to the monitoring of a truck/trailer combination. Figure 5.2 illustrates the sensor network in this scenario. The trailer is equipped with a sensor network and a gateway device. The truck has an on-board unit with an Internet connection via a cellular network.

The sensor network in the trailer can be used for a range of applications, including:

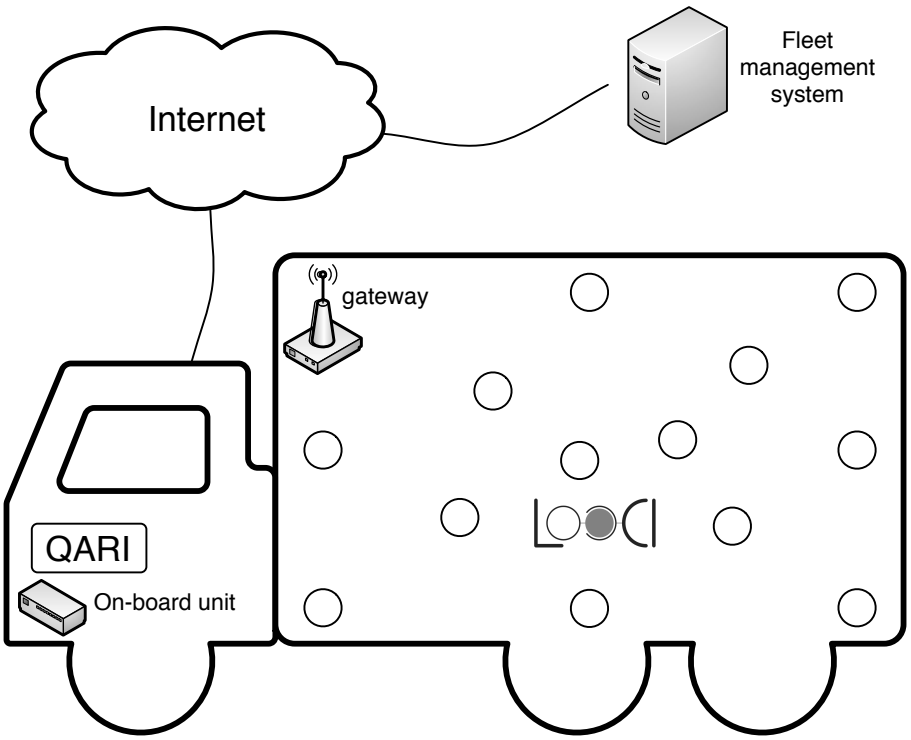


Figure 5.2: Illustration of a sensor network in a truck/trailer combination.

- Status monitoring: during transport, products such as pharmaceuticals or food must be kept within a given temperature range. Sensor networks can provide monitoring of these temperature conditions. When the temperature exceeds the thresholds, the sensor networks trigger alerts to allow early remedy of the problem.
- Intrusion detection: transport of expensive or dangerous goods requires strong physical security to prevent theft and tampering. Sensor networks complement this security with detection of abnormal events such as unauthorized door opening.

The management solutions in this dissertation assist the fleet management system in the back office of the transport company. The requirements for monitoring and intrusion detection depend on the type of cargo. The fleet management system manages the planning of the transports and thus knows what products the trailer

```
<network name="trailer1.A.com" xmin="0" ymin="0" xmax="7"
    ymax="2">
  <node name="gateway.trailer1.A.com" x="0" y="0" range="1"
    accepts="looci:implementation.looci.osgi" />
  <node name="node01.trailer1.A.com" x="0" y="1" range="2"
    accepts="looci:implementation.looci.contiki" />
  ...
  <group name="sensors.trailer1.A.com">
    <member>node01.trailer1.A.com</member>
    ...
  </group>
  <zone name="front.trailer1.A.com" xmin="0" ymin="0"
      xmax="3" ymax="2" />
</network>
```

Listing 5.1: Network description for the truck/trailer scenario.

carries at which point in time. The fleet management system updates QARI's policies according to this information. QARI subsequently updates the software components on the sensor network through a modular runtime system.

The fleet management system submits policies to QARI containing application-level quality requirements. For example, when the trailer is parked under a tree, exposing only half of the trailer to direct sunlight, a difference in the temperature conditions of the front and the back of trailer may occur. It is important to discriminate between these zones in the trailer to allow better framing of the problem. This is expressed in the policies through zones and coverage goals. Another example concerns the correlation between humidity and temperature. To enable the calculation of this correlation, the humidity and temperature measurement should originate from sensors close to each other. QARI's distance goals can express such requirements.

5.1.3 Illustration of the policy specifications

This section illustrates the policy specifications that govern QARI's operation as introduced in Section 4.2 by showing the policies for the truck/trailer scenario of Section 5.1.2. Listing 5.1 shows the network description and Listing 5.2 shows he constraint specification, both provided by the infrastructure administrator. Next, Listing 5.3 shows the composition specification written by the application developer. Finally, Listing 5.4 shows how an application administrator connects the network description and the composition specification through a deployment specification.

```
1  <constraints>
2    <constraint.maxcomponents value="2"
         subject="node01.trailer1.A.com" />
3    ...
4  </constraints>
```

Listing 5.2: Constraint specification for the truck/trailer scenario.

```
1  <composite name="TemperatureMonitoringComposite">
2    <component name="TemperatureSampling">
3      <looci:implementation.looci.contiki
           file="tempsample.comp" />
4      <service name="TemperatureSamplingService">
5        <looci:interface.looci eventtype="101" />
6      </service>
7    </component>
8    <component name="TemperatureAggregation">
9      <looci:implementation.looci.osgi file="tempaggr.jar" />
10     <reference name="TemperatureSamplers"
           multiplicity="0..n">...</reference>
11     <service name="AggregatedTemperatureService">...</service>
12   </component>
13   <component name="TransportMonitoringSystem">...</component>
14   <wire source="TemperatureAggregation/TemperatureSamplers"
15     target="TemperatureSampling/TemperatureSamplingService" />
16   ...
17 </composite>
```

Listing 5.3: Composition specification for a temperature monitoring application in the truck/trailer scenario.

The network description in Listing 5.1 names the network and provides the boundary coordinates for the implicit zone defined by the network, i.e. the physical area contained within the trailer (line 1). Next, the specification contains details about the nodes in the network (lines 2–5): a name, the initial location and the sensing range. The *accepts* property of a node lists the component type that the node supports. Lines 7–10 show the definition of a group containing all sensor nodes (excluding the gateway) in the trailer. The scenario requires the ability to discriminate between different zones in the trailer: line 11 illustrates the definition of a zone representing the front of the trailer. The infrastructure administrator also provides a constraint specification with application independent constraints. Listing 5.2 shows an example that limits the number of components on a node to two.

```
1  <deployment name="TemperatureAndHumidityMonitoringDeployment"
2    composites="ex:TemperatureMonitoringComposite
         ex:HumidityMonitoringComposite">
3    <componentdeployment name="tempsamp"
         component="ex:TemperatureSampling">
4      <target name="sensors.trailer1.A.com">
5        <goal.coverage area="trailer1.A.com" min="1" pref="2" />
6      </target>
7    </componentdeployment>
8    <componentdeployment component="ex:HumiditySampling">
9      <target name="sensors.trailer1.A.com">
10       <goal.coverage area="front.trailer1.A.com" min="0"
             pref="1" />
11       <goal.distance nominal="1" max="2" to="ex:tempsamp" />
12     </target>
13   </componentdeployment>
14   <componentdeployment component="ex:TemperatureAggregation">
15     <target name="gateway.trailer1.A.com" />
16     <dependency element="AggregatedTemperatureService"
           uri="looci://tms.A.com/*/102" />
17   </componentdeployment>
18   ...
19 </deployment>
```

Listing 5.4: Deployment specification for a temperature monitoring application and a humidity monitoring application in the truck/trailer scenario.

The composition specification describes the assembly of an application from it constituent components. The composition specification adheres to the Service Component Architecture (SCA) Assembly specification [110] format. It instantiates the predefined extension points of the SCA assembly specification format to support the description of LooCI compositions. Listing 5.3 shows the (shortened) composition specification for a temperature monitoring application. The application consists of a temperature sampling component (lines 2–7), a temperature aggregation component (lines 8–12) and a transport monitoring system that is part of the fleet management system (line 13). A component definition contains a description of the implementation of the component (e.g. line 3). Additionally, it may also contain services (e.g. lines 4–6) and references (e.g. line 10). A service is equivalent to a provided interface in LooCI, a reference is a receptacle in LooCI terms. Lines 14–15 illustrate the wire connecting the temperature sampling component and the temperature aggregation component.

When a cargo needs, for example, temperature and humidity monitoring, the fleet management system submits a deployment specification to QARI. This deployment specification (Listing 5.4) describes how the compositions for these

two applications should be deployed on the sensor network in the trailer. The specification consists of component deployments (see Section 4.2.4) that map a component to a target. Consider the component deployment on lines 8–13 that maps the humidity sampling component to the sensor nodes in the trailer. It attaches two goals to this mapping: a coverage goal and a distance goal. The coverage goal expresses that the area contained within zone `front.trailer1.A.com` should be covered once if possible (minimum coverage is 0). The distance goal restricts the distance to the temperature sampling component to enable correlation of the measurements. The dependency on line 16 allows to express a dependency on a component managed outside QARI. In this case, the aggregated temperature measurements are sent to the transport monitoring system, a subsystem of the fleet management system. QARI needs this dependency information to create the wire to this external component.

5.2 DAViM implementation

This section discusses the prototype implementation of DAViM. Section 5.2.1 first illustrates the use of DAViM's flexible support for reconfiguration in the warehouse monitoring scenario of Section 5.1.2. Section 5.2.2 presents more details on the prototype and discusses the evaluation of DAViM. This evaluation shows that DAViM has sufficient resource efficiency to run on multi-purpose sensor networks.

5.2.1 Illustration

This section illustrates the use of DAViM's flexible reconfiguration support in multi-purpose sensor networks. Consider the warehouse monitoring scenario of Section 5.1.2. Initially, the sensor network runs a temperature monitoring application for the HVAC system. The temperature monitoring application alerts the HVAC system whenever the temperature exceeds certain thresholds. The appropriate thresholds depend on the type of goods stored in the warehouse. The HVAC system therefore uses DAViM's lightweight script updates to change the thresholds dynamically.

Later on, the warehouse operator deploys a newer version of the HVAC system. To increase the efficiency of the HVAC system in scenarios where different goods are stored in the warehouse, this new version can control the temperature in different parts of the warehouse independently. Therefore, the HVAC system needs information on the location of the temperature alerts to determine the appropriate reaction. To update the temperature monitoring application for this new requirement, the administrator first deploys a new operation library with a localization service. Subsequently, the administrator updates the description of the

virtual machine that hosts the temperature monitoring. The updated description includes the new localization service in the instruction set of the virtual machine. Finally, the administrator updates the application components of the temperature monitoring application to a version that includes the location of the node in the temperature alerts.

DAViM's multiple virtual machine feature supports isolation of different applications. Consider an application for monitoring safety perimeters (see Section 5.1.2). This application requires distance measurements, an operation included with the localization service that the administrator deployed earlier. However, the safety monitoring application should operate independently from the temperature sampling application. Therefore the administrator creates a new virtual machine on the sensor network to host the safety monitoring application. This new virtual machine includes the operations of the localization service in its instruction set. Although both virtual machines share an operation library, DAViM ensures the independent operation of the two applications.

5.2.2 Implementation and evaluation

This section discusses the prototype implementation of DAViM on top of the SOS operating system [57]. It first details the evaluation setup and then presents the evaluation of DAViM.

Prototype implementation. The prototype of DAViM is implemented on micaZ sensor nodes. The micaZ sensor nodes are an example of a low-end sensor node platform. This platform provides an 8-bit AVR ATmega128L MCU, 4 kB of RAM and 128 kB of flash memory for code. DAViM reuses the support for dynamic memory and loadable modules of the underlying operating system. DAViM is implemented on top of the SOS operating system [57], which provides these features and supports the micaZ platform. DAViM's core architecture and the operation libraries are implemented as SOS modules enabling the operation libraries to be loaded dynamically. DAViM's architecture does not specify a specific algorithm for the Coordinator component. The prototype implementation uses the mechanism provided by SOS for distributing operation libraries (SOS uses a variant of MOAP [126]) and Trickle [93] for distributing applications and VM descriptions.

Evaluation setup. The evaluation of DAViM used the avrora AVR simulator [134, 123] to simulate a sensor network with 16 micaZ nodes on a 4x4 grid with 15 m spacing. DAViM is compared with Bombilla, the standard virtual machine included with Maté [90, 91]. Maté pioneered the script-based approach to sensor

network reconfiguration, providing a virtual machine on top of TinyOS [92]. The evaluation used the following versions of the tools: avrora CVS checkout of April 5th, 2007 at 10:25 CEST, SOS 2.0.0, TinyOS 1.1.15 and Maté 2.2.2.

Evaluation. The evaluation of DAViM's architecture and its prototype implementation seeks answers to three questions: (1) does the dynamic nature of DAViM imply an unreasonable overhead compared to existing virtual machine architectures, (2) what is the benefit of dynamic deployment of operation libraries and (3) what is the overhead of the support for multiple virtual machines?

To answer the first question, the execution overhead of DAViM with respect to SOS is measured and compared to the overhead of Bombilla with respect to TinyOS. The evaluation simulated a light sampling application that sends a light sample to a base station every 10 s. The simulations ran for 18000 s (5 hours). The percentage of time the CPU was active was measured for the four implementations. Table 5.1(a) shows that the overhead of DAViM compared to SOS is in the same range of the overhead of Bombilla compared to TinyOS (10-12%). Note that the DAViM implementation is an unoptimized prototype: a quick optimization specialized for the light sampling application reduces DAViM's overhead to 8.9%. To measure DAViM's overhead during dissemination and installation of new application components (scripts), the simulation of the light sampling application was repeated, but a script was injected that restarted the application every 305 seconds. An equivalent experiment was performed with Bombilla. Table 5.1(a) shows again that the overhead is comparable.

To answer the second question of the evaluation, a new instruction was added to both DAViM and Bombilla and the size of the update that the sensor nodes need to disseminate in the network was compared. The new instruction calculates an exponentially weighted moving average (EWMA). To deploy this new instruction for Bombilla, the whole image needs to be replaced using Deluge [66], the over-the-air reprogramming system for TinyOS. The size of this image for micaZ was 50046 bytes. To deploy this new instruction for DAViM, DAViM only needs to load a new operation library. To do this, the module loading system of SOS sent only 612 bytes (two orders of magnitude less). As communication over the radio is a large source of energy consumption, updating the instruction set of a virtual machine is less energy consuming with DAViM than with Bombilla.

To answer the third question, an evaluation was performed of the overhead of DAViM's support for multiple virtual machines. Table 5.1(b) shows the extra size of the virtual machine data structures and the context (i.e. the execution environment for an application component) data structures due to the support for multiple virtual machines. Table 5.1(b) also shows the memory overhead of the extra instance of the Trickle algorithm [93] used to maintain version coherence of the virtual machines. Furthermore, the table contains the size of the packets sent

(a) Average CPU active time of the surge application on different platforms.

	%Active	Overhead
SOS	7.89%	
DAViM	8.80%	11.5%
TinyOS	5.31%	
Bombilla	5.86%	10.3%
DAViM (light sampling optimization)	8.59%	8.9%
DAViM (periodic updates)	8.81%	11.6%
Bombilla (periodic updates)	5.86%	10.3%
DAViM (static mapping)	8.80%	11.50%

(b) Overhead to support multiple virtual machines: (1) extra size of data structures, (2) size of extra network packets and (3) memory due to support for multiple virtual machines compared to total memory for the application(s).

Data structure	**Overhead**
State/VM	25 bytes (60%)
State/context	5 bytes (11%)
Trickle state	27 bytes (36%)
Network packet	**Size**
Trickle VM summary	32 bytes/packet
Trickle VM description	18 bytes/packet
Application(s)	**Overhead**
Light sampling application	87 bytes (31%)
3 complex applications	242 bytes (5%)

Table 5.1: Evaluation of DAViM

by the algorithm. However, the network overhead of Trickle in a stable state is very low [93]. For the light sampling application, 31% of the total memory consumption is related to the support for multiple virtual machines (Table 5.1(b)). However, the application components for this application are very small (6.7 bytes on average). For a more realistic usage scenario where three applications run in their own virtual machine with on average 8 application components of 128 bytes per application, only 5% of the total memory consumption can be attributed to the support for multiple virtual machines. The dynamic mapping of instructions to operations introduces an extra indirection, with an increase in interpretation overhead as a consequence. As can be seen from Table 5.1(a), for the light sampling application, this has no significant effect on the CPU active time (second row: 8.80% active) compared to DAViM with a static mapping (last row: 8.80% active).

These experiments show that DAViM is sufficiently lightweight to operate on

resource constrained sensor networks. The overhead is acceptable given the need for flexible reconfiguration in multi-purpose sensor networks.

5.3 LooCI implementation

This section documents the implementation of the LooCI component model for low-end sensor nodes on top of Contiki [41]. The feasibility of LooCI on high-end sensor nodes has been confirmed through implementations on SunSPOT and OSGi [64], which are not discussed in this dissertation. This section first provides a rationale for the choice for Contiki and LooCI's important design decisions (Section 5.3.1). The text then details how a LooCI component is developed (Section 5.3.2), followed by an explanation of the reconfiguration of a LooCI enabled sensor node (Section 5.3.3). Finally, the inner workings of the implementation are discussed and the measurements that confirm the resource efficiency of LooCI are presented (Section 5.3.4).

5.3.1 Rationale

The goal was of course to provide an implementation of the complete LooCI component model on low-end sensor nodes. The LooCI implementation on Contiki thus provides support for reconfigurable components and LooCI's loosely coupled, distributed, event-based interaction paradigm. LooCI is implemented on top of Contiki for three reasons: (1) it is widely used and has an active community supporting it, (2) it supports a broad range of low-end sensor devices, and (3) it features a (modular) dynamic code loading mechanism [39]. Because of the last reason, Contiki has an advantage over TinyOS [92]. TinyOS is the de facto standard for sensor nodes, but implementing dynamically reconfigurable components on top of TinyOS' statically optimized runtime would be a lot harder.

To provide a gentle learning curve for Contiki programmers, LooCI provides a programming experience as close as possible to the regular Contiki experience. Therefore LooCI's concepts are modeled similar to existing concepts of Contiki. For features not included in the model, the existing Contiki support can be used from within a LooCI component.

The LooCI runtime reuses as much functionality as possible from the Contiki operating system that underpins it (see also Section 5.3.4). The rationale for this design decision is twofold: (1) it avoids duplicate effort and (2) LooCI benefits the most from advances in Contiki. Components are implemented as an abstraction layer on top of Contiki's lightweight processes. The dynamic loading of components is built upon the existing Contiki ELF loader [39]. The event dispatching mechanism makes use of the low-level event dispatching mechanism of

```
#include "contiki.h"
#include "looci.h"

#define TEMP_EVENT 101
#define TEMP_TRESHOLD 20

COMPONENT_INTERFACES(temp_filter, TEMP_EVENT);
COMPONENT_RECEPTACLES(temp_filter, TEMP_EVENT);
COMPONENT(temp_filter, "Temperature Filtering");

LOOCI_COMPONENTS(&temp_filter);

COMPONENT_THREAD(temp_filter, ev, data)
{
  COMPONENT_BEGIN();

  static int temp = 0;
  static struct looci_event temp_event;

  while(1) {
    LOOCI_EVENT_RECEIVE(&temp_event);
    if(temp_event.type == TEMP_EVENT) {
      temp = *((int*)temp_event.payload);
      if(temp > TEMP_TRESHOLD) {
        looci_event_publish(TEMP_EVENT, temp_event.payload,
            temp_event.len);
      }
    }
  }

  COMPONENT_END();
}
```

Listing 5.5: Example LooCI component for temperature filtering.

the Contiki event kernel. For communication LooCI leverages on the uIPv6 stack provided with Contiki.

Finally, to meet the resource efficiency requirement, attention was paid to making LooCI as lightweight as possible. Clearly, the reuse of existing Contiki functionality helps in achieving this goal.

5.3.2 Developing a LooCI component

Listing 5.5 shows the source code (in C) of a LooCI component for temperature filtering. The component accepts temperature events and when the temperature is above a given threshold, the component re-publishes the event. The component thus has one interface and one receptacle of the same type. These interfaces and receptacles, together with the component itself, are declared in lines 7 to 9. These declarations give the LooCI runtime access to the component metadata. The second part of the component is the component implementation (lines 13–31). The component implementation contains the application logic for the component. For the convenience of the programmer, the structure of the component implementation resembles that of a Contiki process. LooCI components interact by publishing events to and receiving events from LooCI's distributed event bus. Line 21 shows how the component receives a temperature event from the bus. When the temperature is above the threshold, the component forwards the event by publishing it to the event bus (line 25).

The following paragraphs provide a sequential walk-through of the source code in Listing 5.5:

Lines 1–5 Similar to normal C programs, a LooCI component starts with including the necessary header files and defining constants using preprocessor macros. The component includes both the Contiki and the LooCI header file. In this example, the component defines a constant for the event type and for the sampling rate.

Lines 7–9 These lines declare the component and its interfaces and receptacles. The declaration of the interfaces and receptacles has to be done before the declaration of the component itself due to limitations of the C preprocessor. The first parameter of the three declarations (`temp_filter`) is the variable name for the structure allocated for the component. The interface and receptacle declarations accept an unspecified number of extra arguments indicating the produced or consumed event types. The declaration of the component includes a string argument that defines the component type.

Line 11 This declaration informs the LooCI runtime of the components in this source file. Although it is possible to declare multiple components in the same file, doing so is discouraged for dynamically loaded components. LooCI discourages this practice because the deployment API (see Listing 3.1) will only return the component identifier for the first component. The LooCI implementation on Contiki also allows components to be statically compiled into the runtime, in which case multiple components per source file causes no problems. When future versions of the LooCI model introduce a separation between deployment and instantiation into the component life-cycle (see

Section 3.4.2), deploying multiple components simultaneously will be fine for dynamic component deployment too.

Lines 13–15 and 30–31 These lines show the basic structure of every component. This structure defines a Contiki process in which the component will execute. The first parameter to `COMPONENT_THREAD` is again the variable name for the component structure. The second parameter is the low-level Contiki event that caused scheduling of the Contiki process. Additional data is passed with this low-level event through the third parameter. Such a low-level Contiki event is distinct from a LooCI event. It gives the component access to the features Contiki provides to processes, such as timers and sensor access.

Lines 17–18 Local variables of a component have to be declared static. This is needed because of the way Contiki processes are implemented.

Line 21 The component will wait on this line until a LooCI event is received. After the event is received, it will be accessible to the component in the `temp_event` variable. A variant of this macro (`LOOCI_EVENT_RECEIVE_UNTIL( event, condition)`) is available that blocks the component until a LooCI event is received or until a given condition is true.

Line 22 The component can access the type of the received event through the `type` field of the `struct looci_event` that contains the received event.

Line 23 The payload of an event can be accessed via the `payload` field of the `struct looci_event`.

Line 25 A component publishes an event with `looci_event_publish(type, payload, length)`. The first parameter is the LooCI event type. The second and third parameter are the payload and the length of this payload. In the example, the component forwards the previously received event and thus passes the payload of that event and its length to this function. As the example illustrates, the length of the event payload is available through the `len` field of the `struct looci_event`.

This example shows that developing a LooCI component requires only a limited overhead in terms of source lines of code. Only 4 lines of code are required for the necessary declarations (lines 7–9 and line 11). The implementation of a component is very similar to the implementation of a Contiki process and requires the same amount of boilerplate code (lines 13–15 and 30–31).

Exhaustive API documentation is included in Appendix B.

```
1  $ looci_shell deploy temp-sample.comp
       2001:db8::11:22ff:fe33:4455
2  2
3  $ looci_shell deploy temp-filter.comp
       2001:db8::11:22ff:fe33:4455
4  3
5  $ looci_shell wireLocal 101 2 101 3
       2001:db8::11:22ff:fe33:4455
6  $ looci_shell wireTo 101 3 2001:db8::11:22ff:fe33:4455
       2001:db8::12:13ff:fe14:1516
7  $ looci_shell activate 2 2001:db8::11:22ff:fe33:4455
8  $ looci_shell activate 3 2001:db8::11:22ff:fe33:4455
9  $ looci_shell getState 2 2001:db8::11:22ff:fe33:4455
10 active
11 $ looci_shell getLocalWires 101 2 2001:db8::11:22ff:fe33:4455
12 3
13 $ looci_shell getOutgoingRemoteWires 101 3
       2001:db8::11:22ff:fe33:4455
14 2001:db8::12:13ff:fe14:1516
```

Listing 5.6: Example of using the shell wrapper around the reconfiguration API to manage a LooCI node.

5.3.3 Reconfiguration of LooCI nodes

The LooCI reconfiguration API (consisting of a deployment, runtime control and introspection API, see Section 3.4.2) enables reconfiguration of a LooCI network from a computer outside the network. For interaction with the LooCI implementation on Contiki, an implementation of these APIs in standard C is provided. In addition, a shell wrapper allows to interact with LooCI nodes from a command line. A Java wrapper enables integration with QARI (see Section 5.5).

Listing 5.6 shows how to use the shell wrapper for the reconfiguration API to deploy a temperature sampling and a filter component to a LooCI node. The example first shows how to deploy the two components to a LooCI node (lines 1–4). Then the temperature events from the sampling component are wired to the filter component (line 5). The events produced by the filter component are then wired to a remote node (line 6). Next, both components are activated (lines 7 and 8). Lines 9 to 13 show how to introspect the state of the nodes afterwards.

A sequential walk-through with more details about the commands in Listing 5.6 is now presented:

Lines 1–4 show how to deploy the temperature sampling and temperature filtering components to the node with node identifier `2001:db8::11:22ff:fe33:4455`

(an IPv6 address in this case). The `deploy` command returns the component identifier that the runtime assigned to the new component instance.

Line 5 The `wireLocal` command wires two components on the same node. The example uses the command to wire the interface with event type 101 of the temperature sampling component to the receptacle with event type 101 of the temperature filtering component. The event type of the interface and the event type of the receptacle should be compatible (see Section 3.4.1).

Line 6 wires the interface with event type 101 of the temperature filtering component to the remote node with identifier `2001:db8::12:13ff:fe14:1516`.

Lines 7–8 After deployment, the components are in the deactivated state by default. With these commands, the two components are activated.

Lines 9–13 show some of the introspection capabilities of LooCI. With the command on line 9, one verifies that the temperature sampling component is now active. Line 11 queries for the local components connected to interface 101 of the temperature sampling component. Finally, line 13 shows how to query for the nodes that receive the events produced by the temperature filtering component.

The full API documentation, including documentation on the usage of the shell wrapper, is included in Appendix C.

5.3.4 Implementation and evaluation

The implementation of the LooCI runtime on top of Contiki leverages several features of Contiki to minimize the footprint of the LooCI runtime. This section describes how the implementation built upon Contiki. It then briefly discusses the hardware platform used. The discussion continues with the result of the measurements performed to assess the resource efficiency of LooCI. These measurements were conducted using Contiki 2.4.

Leveraging Contiki. Figure 5.3 shows the major subsystems of the LooCI runtime and their interactions. The LooCI runtime leverages upon several features of Contiki: (1) the reconfiguration engine builds upon Contiki processes and the Contiki ELF loader, (2) the networking framework uses the uIPv6 communication stack of Contiki and (3) the distributed event bus leverages Contiki's low-level event kernel.

The LooCI reconfiguration engine manages the life-cycle of the LooCI components and reconfigures the wires in the distributed event bus. A LooCI component is

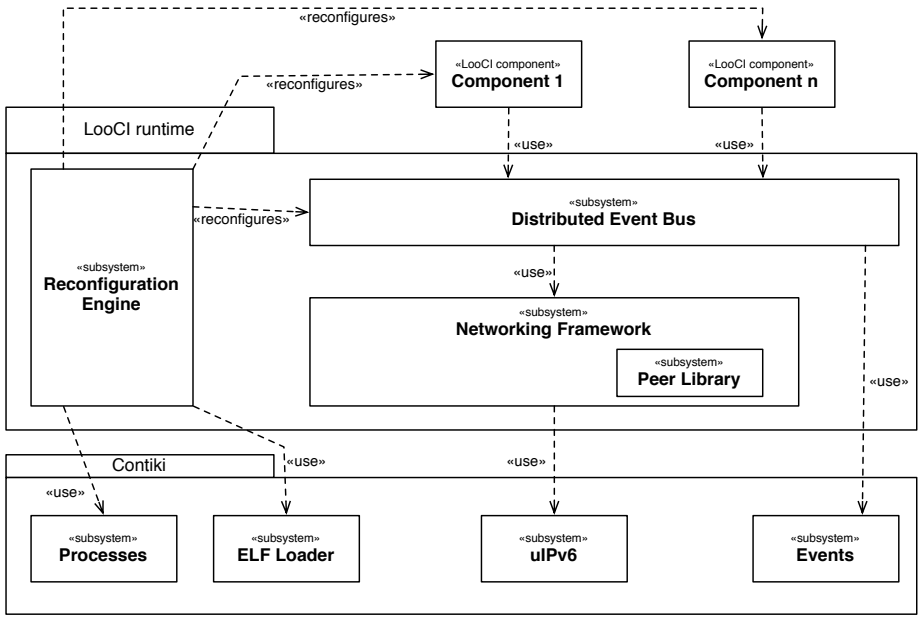


Figure 5.3: Overview of the LooCI runtime implementation on Contiki.

implemented as an abstraction layer on top of Contiki's lightweight processes: the component implementation executes within a Contiki process and the component data structure wraps a Contiki process structure to add extra metadata (component identifier, component state, interfaces and receptacles). The reconfiguration engine reuses the Contiki support for starting and stopping processes to implement activation and deactivation of components.

The LooCI reconfiguration engine builds upon the Contiki ELF loader [39] to enable dynamic deployment of components. The LooCI implementation extended the Contiki ELF loader with support for loading and unloading multiple ELF files. Since recent Contiki code bases do not include a ready-to-use code distribution mechanism, the LooCI runtime implements simple unicast TCP-based code distribution. There is certainly room for improvement in this code distribution mechanism, but since this work does not focus on code distribution, this implementation is considered sufficient for the prototype.

LooCI's distributed event bus implements the support for the distributed event-based interaction paradigm. To fully decouple the component producing an event from the component receiving it, the event publishing and delivery must be asynchronous. The Contiki kernel is inherently event-based and provides an

	Contiki	LooCI	Overhead	Overhead (% of total)
Flash (ROM)	40614 (31%)	65012 (50%)	24398 (+60%)	19%
RAM	8885 (54%)	10083 (62%)	1198 (+13.5%)	7%

Table 5.2: Flash and RAM size in bytes of a temperature sampling application in Contiki and LooCI.

asynchronous event mechanism LooCI can built upon. The `looci_event_publish` function queues a low-level Contiki event for delivery to the LooCI event bus and then returns immediately to the calling component. The reception of a LooCI event is also an asynchronous operation based on the low-level Contiki event system. To provide a similar programming experience as with Contiki processes, LooCI provides a blocking (`LOOCI_EVENT_RECEIVE`) and a non-blocking (`LOOCI_EVENT_RECEIVE_UNTIL`) macro to access this operation.

The networking framework abstracts the details of network communication for the other subsystems in the LooCI runtime. This loosens the dependency of the LooCI runtime on the specific network technology. The networking framework currently uses the uIPv6 network stack of Contiki. The peer library, which is part of the networking framework, abstracts the specifics of the addressing scheme by replacing node addresses with a local identifier. This has the additional advantage that each node address (a 16 byte IPv6 address in this case) has to be stored only once. The networking framework abstracts the mechanisms for sending and receiving network packets by providing an interface for sending events and by dispatching incoming events to a callback function in the event bus. Currently, the networking framework uses UDP unicast communication to send the events. For implementation of the `wireToAll` functionality, it uses IPv6 multicast to the *all link-local nodes* multicast address (`ff02::1`).

Hardware platform. The LooCI runtime was developed and tested on AVR Raven nodes. These nodes have an ATmega1284P 20MHz 8-bit microcontroller, 16 kB RAM and 128 kB flash memory for code. This platform was chosen for the prototype because it has representative specifications for the low-end class of devices and because Contiki has good support for it. Both the Contiki networking stack and the Contiki ELF loader are implemented on this platform.

Measurements. The LooCI runtime combined with a sample and send temperature component consumes 65012 bytes of flash memory (see Table 5.2). A functionally equivalent Contiki application requires 40614 bytes of flash memory. The overhead of the LooCI runtime is thus 24398 bytes, which is 19% of the total flash memory available. The symbol table needed for the ELF loader makes

up about 10 kB (10074 bytes for a blank LooCI runtime) of this overhead. In the symbol tables lies an opportunity for future optimization, since it currently includes symbols for low-level functionality that shouldn't be used directly from LooCI components. Removing these symbols from the symbol table could reduce the overhead thereof.

The RAM required for the LooCI runtime, including the sample and send component, is 11107 bytes. The preallocated heap for the RAM of dynamically loaded components consumes 1024 bytes thereof. The equivalent Contiki application requires 8885 bytes of RAM. The LooCI runtime thus has an overhead of 1198 bytes of RAM (excluding the preallocated heap). This is 7% of the total RAM available.

Over-the-air deployment of the temperature sampling component requires on average (10 samples) 11.7 s (standard deviation 1.5 s). Since Contiki's uIPv6 stack on AVR Raven does not support multi-hop networking in version 2.4, this experiment was conducted in a single-hop network. The file size of this component is 1720 bytes. Only 231 bytes of these 1720 bytes are useful content. The rest is due to the large overhead of the ELF file format. Dunkels et al. [39] also describe this problem and they propose a solution in the form of CELF. CELF eliminates a lot of the overhead of ELF by replacing the 32-bit data types with 8-bit and 16-bit data types. Using CELF would significantly reduce the overhead and the time for sending a component to a LooCI node. However, Contiki contains no implementation of this optimization for the AVR platform. Since code distribution is not the focus of this work, this optimization opportunity was not investigated further.

To conclude, the LooCI runtime has an overhead of 19% of the flash memory and 7% of the RAM. Dynamic deployment of a component (1720 bytes) requires on average 11.7 s. Considering the extra features provided by LooCI and the unoptimized implementation of the LooCI runtime, this shows the feasibility of LooCI for resource constrained sensor nodes.

5.4 QARI implementation

This section details the prototype implementation of QARI on top of the OSGi platform. Section 5.4.1 provides a rationale for the technological choices. Section 5.4.2 illustrates the use of QARI in both the warehouse monitoring and truck/trailer monitoring applications (see Section 5.1.2). Finally, Section 5.4.3 discusses implementation details of QARI and provides the results of the measurements that confirm QARI's feasibility and fitness for managing distributed applications on multi-purpose sensor network.

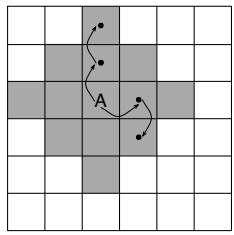


Figure 5.4: Area covered by node A with a sensing range of 2.

5.4.1 Rationale

The prototype implementation of QARI is built on top of the OSGi platform. The OSGi platform is widely used for a variety of applications, including gateway applications [114]. OSGi features a modular runtime for Java-based applications. It is thus a good fit for the implementation of QARI's modular architecture.

The planning problem in QARI is a multi-objective combinatorial optimization problem (see Section 4.5.1). The prototype implementation of QARI leverages the Drools Planner framework [78] to provide a planning strategy based on local search. Drools Planner is part of the JBoss Drools [77] project and builds upon the rule engine of Drools for calculation of the score (utility) of a solution. Drools Planner's Java-based nature facilitates integration with OSGi applications such as QARI.

5.4.2 Illustration

This section illustrates both planning strategies introduced in Section 4.5.1. The algorithm based on a problem specific heuristic (see Figure 4.8) is applied to the warehouse monitoring scenario in Section 5.1.2. The illustration of the truck/trailer scenario uses the planning strategy based on local search.

Warehouse monitoring. Consider the use of QARI for managing a temperature monitoring application and a humidity monitoring application on the sensor network in a warehouse. Both applications consist of a sampling component on the sensor network that provides the HVAC system with information on the conditions inside the warehouse. QARI's problem specific heuristic is designed to handle coverage requirements only. The experiment uses different coverage requirements for both applications.

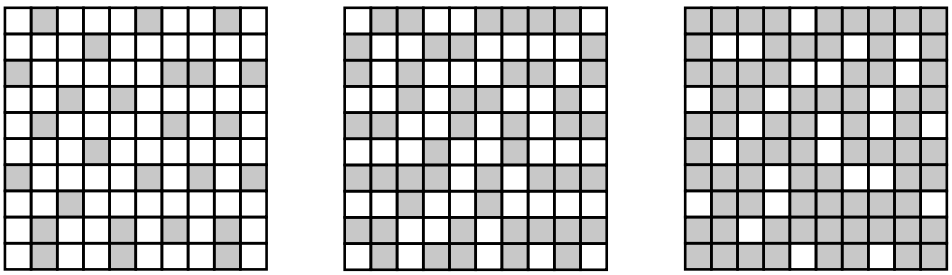


Figure 5.5: Calculated deployment patterns for a single component type using the algorithm in Figure 4.8.

The warehouse spans an area of ten by ten meters with a sensor node in each square meter of the warehouse. For the calculation of the coverage, the area covered by a node is approximated as shown in Figure 5.4. The network is overlaid with a grid and distance is measured on this grid using the Manhattan distance metric. A sensor node then covers the squares less within its sensing range. This example uses a sensing range of 1, thus a sensor node covers the square containing the node and the squares directly surrounding it (left, right, top and bottom).

First, deployment specifications are submitted for the temperature monitoring application only. The experiment is repeated with three different coverage goals. Each coverage goal has a minimal and a preferred (nominal) coverage level. Figure 5.5 shows the deployment patterns generated by QARI for the following coverage requirements (minimal/preferred): 0/1, 1/2, 2/3. The figure includes the minimal and average coverage actually achieved by the deployment pattern. In each case, the minimal coverage satisfies the preferred coverage level specified in the coverage goal.

Next, deployment specifications are submitted for both the temperature and humidity monitoring applications. Both applications target a sampling component to the sensor network. Figure 5.6 shows the result of QARI's planning for several combinations of (preferred) coverage requirements. The algorithm in Figure 4.8 is able to spread the two applications over the nodes of the network. Only when both applications require 2-coverage, the need arises for deploying both sampling components on the same node. In the other cases, there are no nodes that run both components.

Figure 5.7 shows the minimal and average coverage over time for the temperature sampling component in the presence of random node failures (and nodes coming back online). The deployment specification required minimally 1-coverage and preferably 2-coverage for the temperature sampling component. As the figure

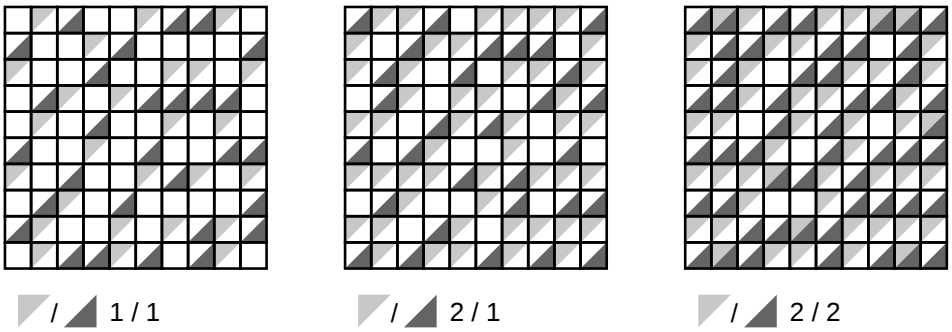


Figure 5.6: Deployment patterns for a shared network as calculated by the algorithm in Figure 4.8: temperature sampling (light gray) and humidity sampling (dark gray). The indicated coverage is the preferred (nominal) value.

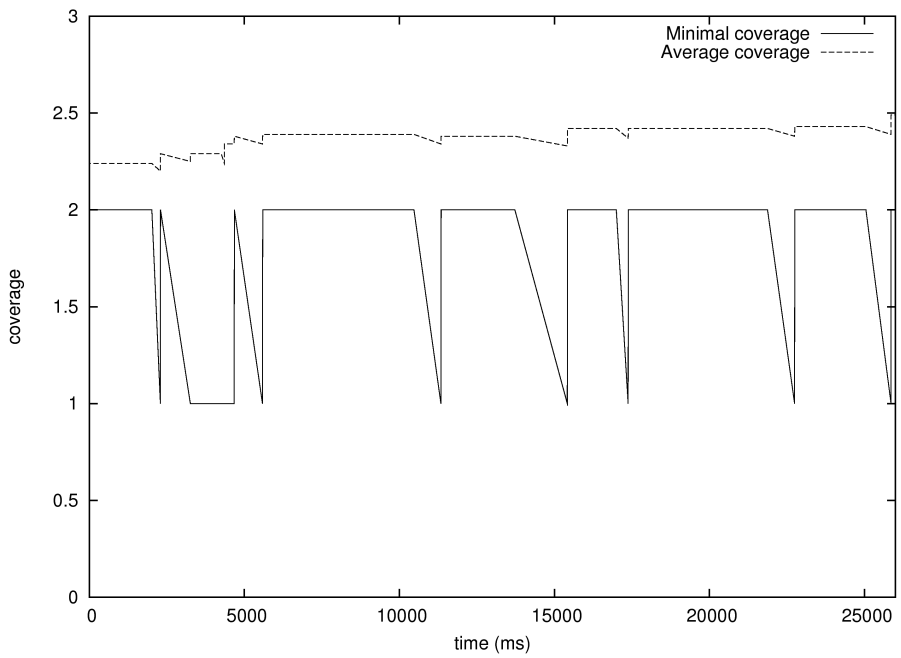


Figure 5.7: Minimal and average coverage over time for the temperature sampling component in the presence of random node failures.

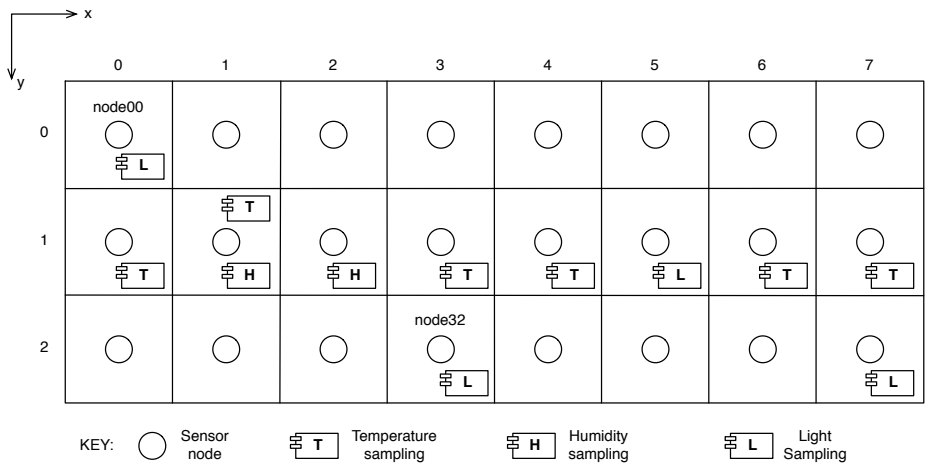


Figure 5.8: Example result of QARI's planning for the components on the trailer in the truck/trailer monitoring scenario.

shows, QARI keeps the minimal coverage between these bounds. At around 3 seconds, the minimal coverage is lower than the preferred coverage for about 1.5 seconds. This is because not enough nodes were available to realize the preferred coverage for some areas of the network. QARI recovers from this situation when enough nodes reappear. Even if it is not possible to meet the requirements, QARI tries to do *as good as possible*. This results in a good level of average coverage despite the fact that the coverage of some areas is lower than preferred.

Truck/trailer monitoring. QARI's planning strategy based on local search is able to incorporate various types of goals and constraints. It is applied to the truck/trailer monitoring scenario in Section 5.1.2. This example illustrates QARI's ability to handle diverse goals and constraints. The example also shows QARI's management console that visualizes information in QARI's information repository.

The trailer of 3 by 8 meter contains a sensor node in each square meter (see Listing 5.1 and Figure 5.8). The sensing range of each node is 2 meters (the same approximation is used for calculating coverage as in the warehouse monitoring scenario above). Now consider a cargo that requires two applications on the sensor network in the trailer: a status monitoring application (temperature and humidity) and an intrusion detection application. Section 5.1.3 introduced simplified input policies for the status monitoring application. Listing 5.4 shows the deployment specification for this application. It requires a temperature sampling component

and a humidity sampling component on the trailer's sensor network. The intrusion detection application needs light sampling on the sensor nodes in the trailer. Thus, QARI must assign the following components to the sensors in the trailer while satisfying the indicated goals:

- Status monitoring application:
  - a temperature sampling component that covers the whole trailer area (minimum: 1-coverage, preferred: 2-coverage)
  - a humidity sampling component that (1) covers the front of the trailer (minimum: none, preferred: 1-coverage) and (2) is deployed close to the temperature sampling component (maximum distance: 2 meters, nominal distance: 1 meter).
- Intrusion detection application:
  - a light sampling component that covers the whole trailer area (minimum: 1-coverage, preferred: 1-coverage).

Apart from the components targeted at the sensor nodes in the trailer, both applications need additional components targeted at specific nodes. The status monitoring application requires aggregation components on the gateway in the trailer and display and alerting components on the on-board unit of the truck. The intrusion detection application includes a door sampling component on the door sensor, an intrusion detection component on the gateway and a display component on the truck's on-board unit.

The experiment first submits the deployment specification for the status monitoring application and later on submits the deployment specification for the intrusion detection application. Figure 5.8 shows an example result of QARI's planning for this scenario after both deployment specifications have been submitted. The planning strategy used is non-deterministic, thus the figure shows only one possible outcome of the planning. It can be verified from the figure that all nominal goal levels are satisfied.

Figures 5.9, 5.10 and 5.11 show screenshots of the QARI management console that contains different views on the data in QARI's information repository. The node view (Figure 5.9) details, for each node, the constraints and the components assigned to that node. A constraint is shown green if it is satisfied and red when a violation occurs. The goal view (Figure 5.10) provides an overview of the goals and targets in the deployment specifications that QARI currently manages. The goals are colored green, orange or red based on the quality level that QARI achieved: when the nominal value of the goal is satisfied, the goal is shown in green; when only the minimal value is met, the goal is shown in orange; when the goal is unsatisfied, the goal is colored red. Finally, the coverage view (Figure 5.11) shows

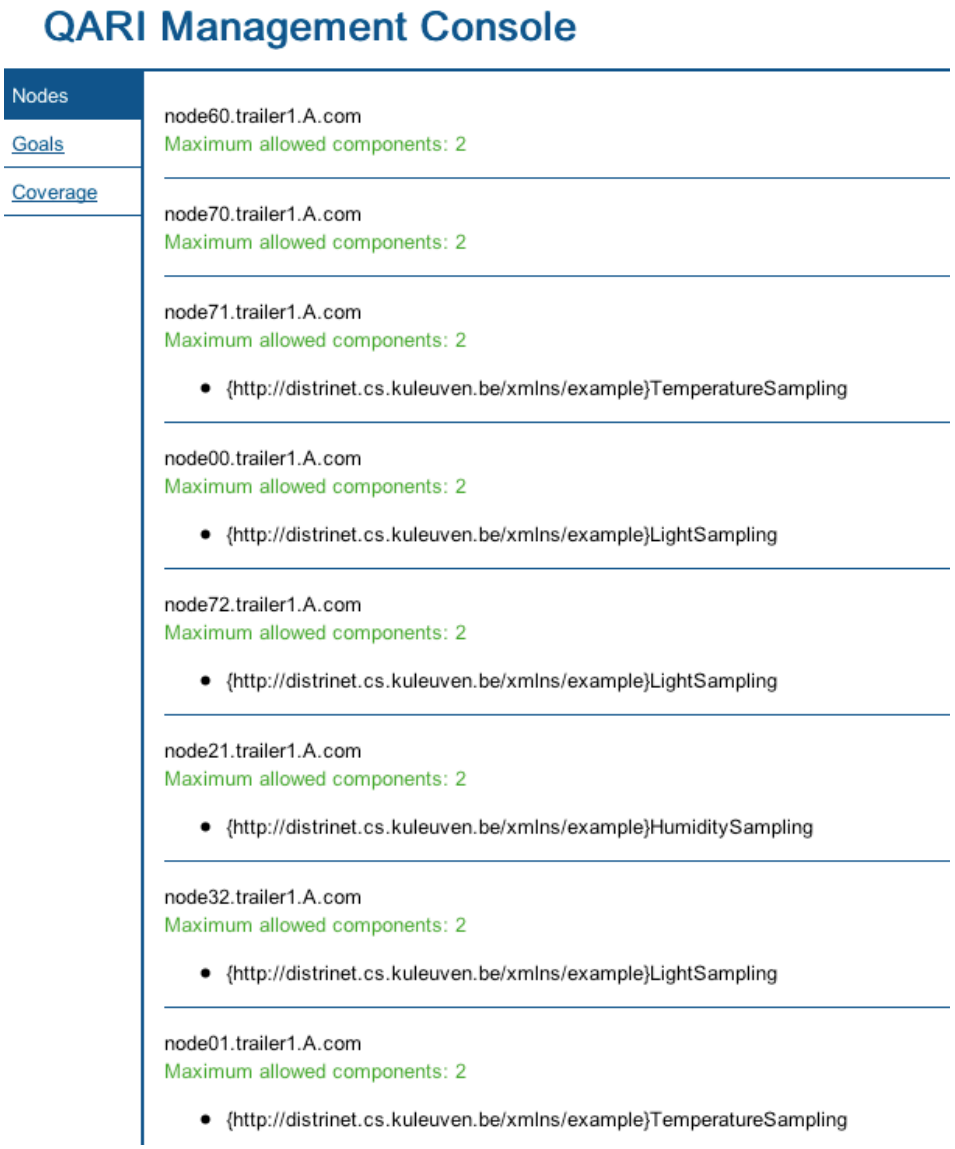


Figure 5.9: QARI Management console: node view.

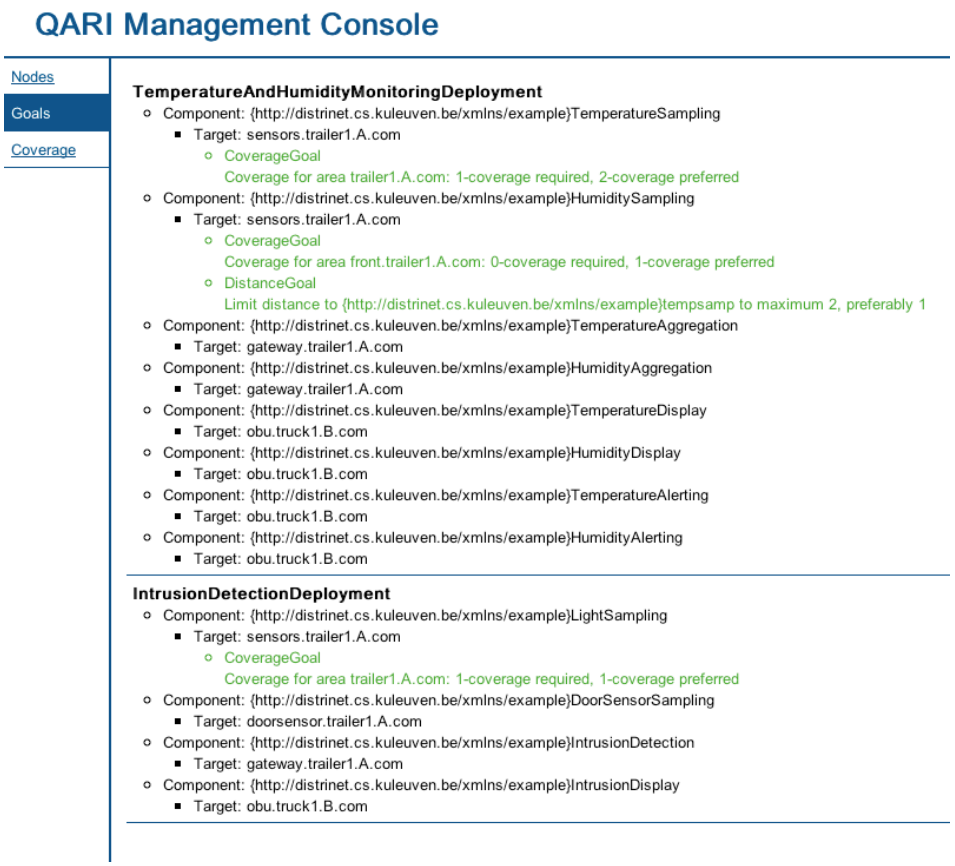


Figure 5.10: QARI Management console: goal view.

the evolution of coverage over time for the components with coverage goals. The graphs show both the minimal and average coverage achieved (updated every 5 s). The drop in coverage near the end of the graphs is due to a node failure introduced during the experiment. The figure shows how QARI recovers from this node failure: the coverage is restored quickly through re-planning.

5.4.3 Implementation and evaluation

This section first details the application of the Drools Planner framework [78] for the implementation of QARI's planning strategy based on local search.

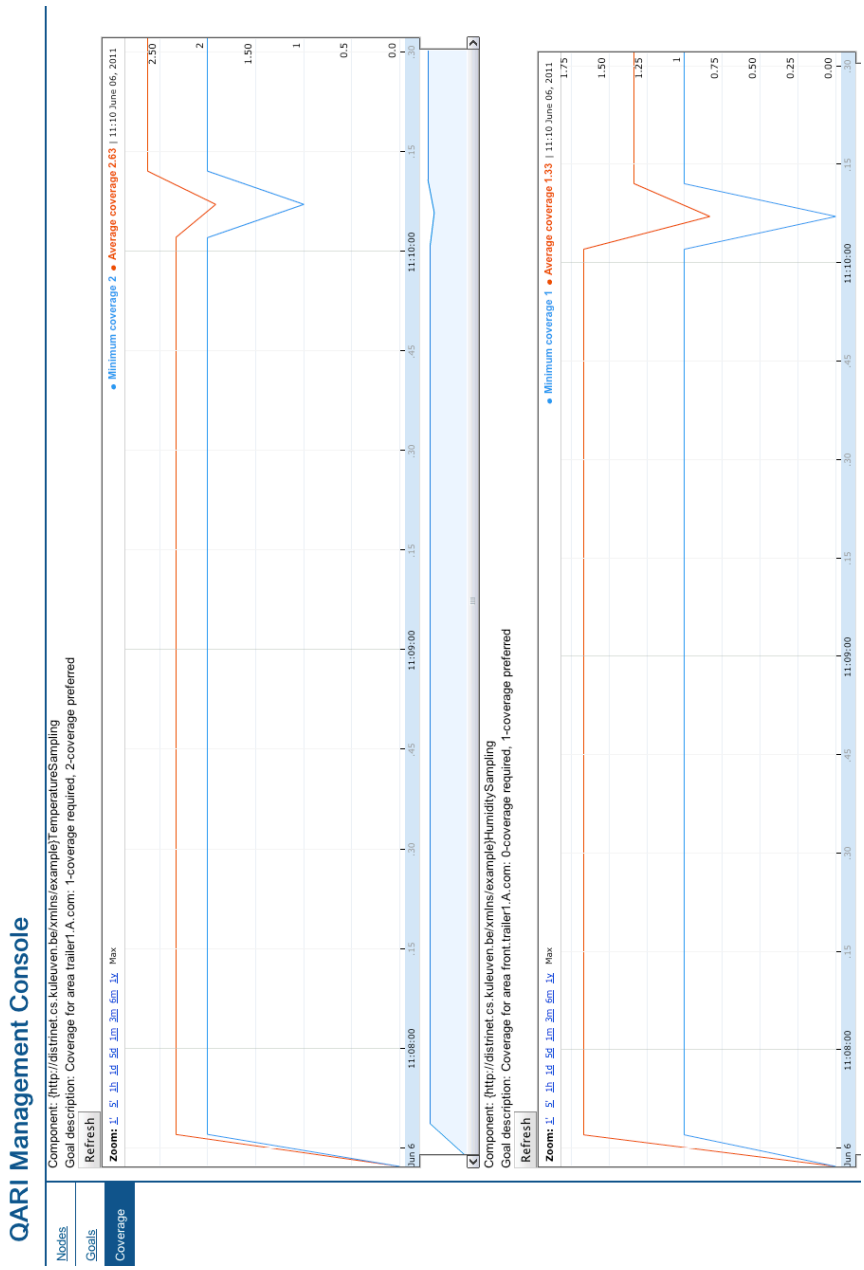


Figure 5.11: QARI Management console: coverage view.

Subsequently, it provides the results of the measurements that were performed to confirm QARI's feasibility and to assess QARI's performance.

Using Drools Planner. The implementation of QARI's local search based planning strategy uses Drools Planner. Drools Planner provides a framework for implementing planning solutions. Drools Planner includes implementations of several local search algorithms, such as tabu search and simulated annealing. Local search algorithms search the solution space by moving from one solution to another guided by a meta-heuristic. To implement a planner using the Drools Planner Framework, one needs to answer three questions: (1) What constitutes a solution? (2) How does one move through the solution space? (3) How to calculate the score (utility) for a solution? The implementation of a planning strategy for QARI answers these questions as follows:

Solution. A solution to the planning problem (Section 4.5.1) is a set of assignments. Each assignment maps a component to a set of nodes.

Moves. To move from one solution to another solution, three types of moves are defined: (1) add a node to an assignment, (2) remove a node from an assignment and (3) switch a node in an assignment with a node not in the assignment.

Scoring. Scoring a solution means computing a utility value for a solution. Drools Planner calculates two scores for each solution: a hard score and a soft score. The hard score must be used for essential optimization criteria, the soft score can be used for optional optimization criteria. Improving the hard score takes precedence over improving the soft score. QARI uses the hard score for satisfying the constraints and for reaching the minimal quality level of a goal. The soft score is used for reaching the nominal quality level of a goal and for system-level optimization criteria. QARI's system-level optimization criteria favor less nodes per assignment, less components per node and less changes to the existing network state.

Hardware platforms. QARI is implemented on top of OSGi and thus runs on each platform that features an OSGi runtime. The measurements below are performed on three different hardware platforms. The warehouse monitoring scenario is evaluated on a linux desktop (Sun Java 1.6.0_16, Pentium D 3.2GHz, 1GB RAM). The evaluation of the truck/trailer scenario uses both a MacBook Pro (Intel Core i5 2.4GHz, 4GB RAM) with Java HotSpot(TM) 64-Bit Server VM (build 1.6.0_22) and a PC Engine ALIX board (AMD Geode LX 500MHz, 256MB RAM) with OpenJDK Client VM (build 1.6.0_0-b11). The former two platforms are standard desktop environments, the latter is representative for a gateway platform or on-board unit in a truck/trailer combination.

	Status monitoring	Intrusion detection	Node failure	Node reappearance
Macbook Pro	4668 (306)	3465 (477)	2089 (485)	1592 (62)
ALIX	57296 (3374)	56815 (8892)	36649 (2286)	28856 (1026)

Table 5.3: Planning time in ms for the truck/trailer monitoring scenario: average over 10 runs (standard deviation).

Measurements. First, the time to plan a solution for the warehouse monitoring scenario with the algorithm in Figure 4.8 was measured. Once the areas in the grid are sorted, the algorithm calculates the assignment in $O(n)$ time, where n is the number of areas. The calculation of 1000 assignments for the 10x10 grid (see Section 5.4.2) requires on average 6233 ms (10 samples; standard deviation 25 ms) on the linux desktop computer introduced above. The time to calculate the assignment is thus orders of magnitude smaller than the time it takes to actually deploy a component to a node (see Section 5.3.4).

Second, the planning times for the truck/trailer monitoring are measured on both the MacBook Pro and ALIX platforms (see Table 5.3). For these experiments, Drools Planner was configured to perform at most 1000 iterations and to stop after 10 iterations without improvement of the solution. Each result is averaged over 10 runs.

Planning the deployment of the status monitoring application (temperature sampling and humidity sampling components) required on average 4668 ms on the MacBook Pro platform. After adding the intrusion detection application (light sampling component), the planning strategy spent on average 3465 ms updating the assignments. Recovering from a node failure required 2089 ms of planning time and re-planning after the failed node reappeared took 1592 ms. The same experiments performed on the ALIX platform yields the results in the second row of Table 5.3. Although the results for the ALIX platform are an order of magnitude larger, these are still acceptable since planning such a deployment manually would require even more time and is a lot more error-prone.

5.5 Integration

This section describes the integration of QARI and LooCI in a small scale testbed of AVR Raven sensor nodes. It describes the testbed setup and reports on the experiments that show QARI's autonomy and QARI's multi-application support.

```
<network name="testbed" xmin="0" ymin="0" xmax="4" ymax="1">
  <node name="raven0" x="1" y="0" range="2"
    accepts="looci:implementation.looci.contiki" />
  <node name="raven1" x="1" y="1" range="2"
    accepts="looci:implementation.looci.contiki" />
  <node name="raven2" x="3" y="0" range="2"
    accepts="looci:implementation.looci.contiki" />
  <node name="raven3" x="3" y="1" range="2"
    accepts="looci:implementation.looci.contiki" />
  <group name="ravens.testbed">
    <member>raven0</member>
    <member>raven1</member>
    <member>raven2</member>
    <member>raven3</member>
  </group>
</network>
```

Listing 5.7: Network specification for the testbed of AVR Raven sensor nodes.

Testbed setup. QARI and LooCI are integrated in a small scale testbed of four AVR Raven nodes. Listing 5.7 shows the network specification for the testbed. The physical area of the testbed is a grid of 5 by 2. The second and fourth column of the grid contain an AVR Raven in each row. The same approximation of coverage (see Figure 5.4) is used again. This means that two nodes (in a different column of the grid) are needed to cover the whole physical area. If 2-coverage is requested, all four nodes are needed. The tests use a constraint specification that puts a limit of two on the number of components on each node. QARI manages the testbed from an ALIX PC Engine gateway (see Section 5.4.3).

For this testbed a simple node failure monitor was implemented using a heartbeat. Each node in the testbed runs a heartbeat component that sends a heartbeat to QARI every 10 s. If three successive heartbeats are missed, QARI considers the node offline. This monitoring strategy is not optimal and causes a certain overhead on the sensor network, however, it suffices for the controlled experiments.

Although this dissertation does not focus on monitoring, monitoring is considered an important area for future work. The scalability of the solution in practice will depend to a large extend on the scalability of the monitoring. Planning and execution of (repair) actions are discrete events triggered by a change in the network or the goals. Monitoring, however, is continuously active in the sensor network. Monitoring solutions that limit overhead will thus be crucial. The monitoring overhead can be decreased, for example, by piggy-backing monitoring information on data traffic, by applying passive monitoring techniques or through on-demand polling via LooCI's introspection API. Of course, there will be a trade-off between the monitoring overhead and the speed of detection of events such as

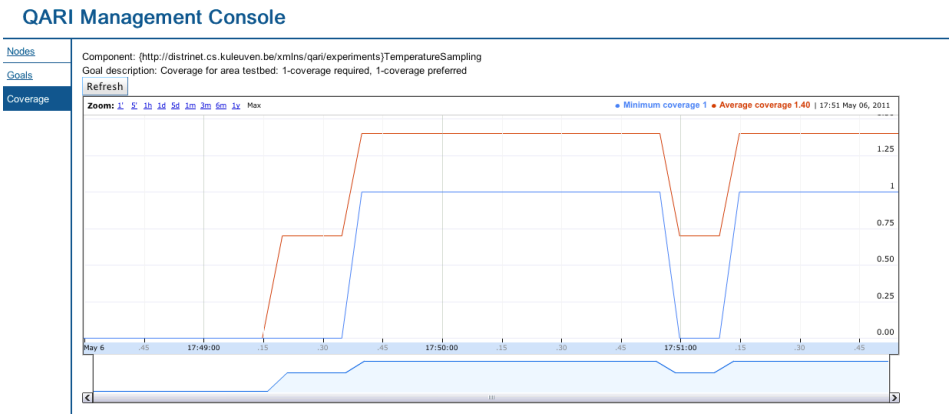


Figure 5.12: Coverage view in the QARI management console for a temperature sampling component on the testbed.

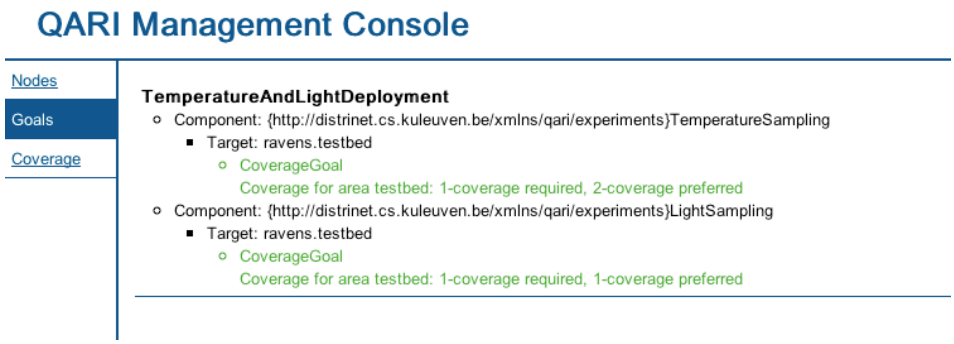
node failures.

Autonomy. Figure 5.12 shows the coverage view in the QARI management console for a temperature sampling component deployed to the testbed with a nominal requirement of 1-coverage and a minimal requirement of 1-coverage. The coverage graph clearly shows the initial deployment phase. QARI is able to autonomously plan and execute the deployment in just over one minute (including the deployment to two Contiki nodes, which takes approximately 24 s, see Section 5.3.4).

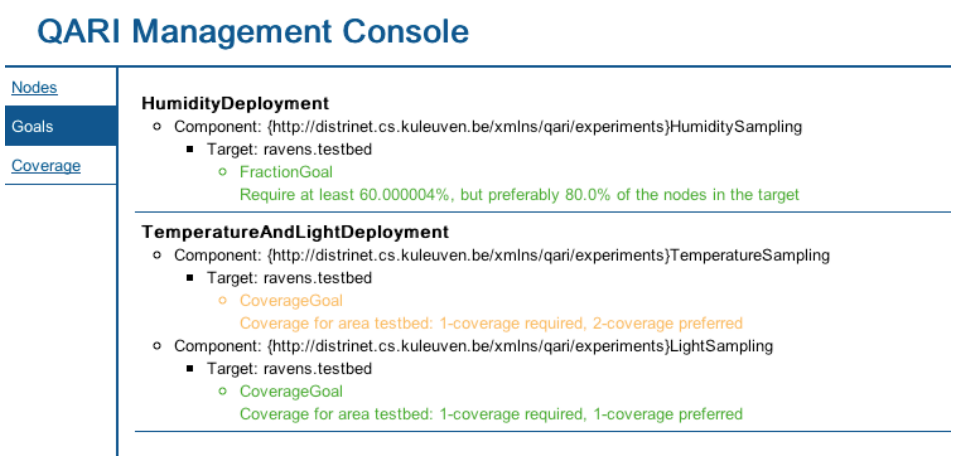
The coverage graph also illustrates QARI's ability to recover from a node failure. The figure shows that QARI recovers from the node failure within approximately 20 s after it was detected. This is a reasonable time frame and (even for this simple setup) considerably faster than a human operator would be able to react.

Multi-application support. QARI handles multiple competing applications by optimizing the overall utility of the network. To illustrate this trade-off, the experiment deployed the following components to the testbed:

- A temperature sampling component with nominal requirement of 2-coverage and minimal requirement of 1-coverage (i.e. a minimum of 2 nodes is required, but 4 nodes are needed for the nominal quality level).



(a) Only temperature sampling and light sampling components.



(b) After adding a humidity sampling component.

Figure 5.13: Goal view in the QARI management console for the multi-application experiment on the testbed.

- A light sampling component with nominal requirement of 1-coverage and minimal requirement of 1-coverage (i.e. 2 nodes).
- A humidity sampling component with nominal requirement of 80% of the nodes and minimal requirement of 60% of the nodes (i.e. minimum 3 nodes, but 4 nodes for the nominal requirement).

Since each node can host at most 2 components (due to the constraint specification), only 8 component instances can be active simultaneously. To satisfy

the nominal quality levels of all three components, 10 component instances are required (4 temperature sampling components, 2 light sampling components and 4 humidity sampling components). QARI thus has to make a trade-off between the applications.

The experiment first submitted a deployment specification for the first two components to QARI. With only the first two components, QARI can satisfy the nominal quality levels of the two components (see Figure 5.13(a)). Next, an additional deployment specification was submitted for the humidity sampling component. At this point, QARI cannot satisfy all nominal quality levels simultaneously. Figure 5.13(b) shows that QARI changed the assignment for the temperature sampling component. This component now only reaches its minimal quality level (the goal is shown in orange).

5.6 Summary

This chapter presented the prototype implementations of DAViM, LooCI and QARI. It first introduced two example scenarios in the context of transport and logistics. These scenarios were then used to illustrate the implementations. The scenarios included the use of multi-purpose sensor networks in warehouse monitoring and in truck/trailer monitoring. The chapter continued with an illustration of the policy specifications for the truck/trailer monitoring scenario.

Next, the chapter introduced the implementation of DAViM on top the SOS sensor operating system. DAViM is a dynamically adaptable virtual machine that supports flexible reconfiguration at different granularities and provides multiple virtual machines to isolate different applications (see Chapter 3). The use of DAViM's features were illustrated in the warehouse monitoring scenario. Section 5.2.2 presented the evaluation of DAViM that shows DAViM's resource efficiency.

Then the text discussed the implementation of LooCI on top of the Contiki sensor operating system. LooCI is a component model for multi-purpose sensor networks featuring runtime reconfigurability and a distributed loosely coupled interaction paradigm (see Chapter 3). The development of a LooCI component was illustrated and examples of the reconfiguration of components and their connections were shown. Section 5.3.4 showed how the implementation leverages the Contiki operating system to assure a lightweight LooCI runtime. Section 5.3.4 concluded with a discussion of the evaluation that confirms the resource efficiency of the LooCI implementation.

The chapter continued with an illustration of the prototype implementation of QARI, a management solution for assembly and deployment of distributed

applications. The use of QARI was illustrated in both the warehouse monitoring and truck/trailer monitoring scenarios. The autonomy and the multi-application support built into the architecture was shown. Measurements of the performance of our prototype implementation confirm the feasibility of running QARI on a gateway device. Finally, the text reported on the integration of QARI and LooCI in a testbed of AVR Raven nodes. The experiments on this testbed show the practical feasibility of the approach.

Chapter 6

Conclusion and future work

This dissertation concludes with a recap of the contributions. Next, Section 6.2 discusses some limitations of the approach that require future work. Finally, Section 6.3 provides an outlook on the longer term future of multi-purpose sensor networks.

6.1 Contributions

This dissertation presented management solutions for distributed applications in multi-purpose sensor networks. Multi-purpose sensor networks represent an emerging use case of wireless sensor networks. Multi-purpose sensor networks no longer serve a single purpose, but are a reusable asset that host multiple applications for a variety of actors. Multi-purpose sensor networks increase the return on investment of wireless sensor networks, given that the operational expenses are kept under control. Automated management solutions are an important tool to reduce these operational expenses.

Management solutions for multi-purpose sensor networks must meet four requirements: (1) resource efficiency, (2) autonomy, (3) reconfigurability and (4) multi-application support. The first requirement stems from the resource constraints of wireless sensor nodes. Management solutions must take these constraints into account. Second, to avoid the need for manual intervention due to the inherent dynamism of sensor networks, the management solutions must autonomously handle network changes. Third, to handle these changes and accommodate the evolving end-user requirements, support for flexible reconfiguration is indispensable. Finally, multi-application support is essential to facilitate true multi-purpose usage of the sensor network.

To address resource efficiency, the research in this dissertation leverages the resources in the gateway and back-end tiers that interact with the multi-purpose sensor network. The presented solutions facilitate autonomy and ease multi-application support by raising the abstraction level of the policies that govern the management solutions. Modularization of the software on the sensor nodes is used to improve reconfigurability and to enable multi-application support through distributed application assembly.

Chapter 3 presented two contributions to the state-of-the-art in runtime support for reconfigurability. Both contributions provide flexible reconfiguration and a foundation for multi-application support through modularity. The first contribution, an adaptable virtual machine (DAViM), focuses on maximizing the modularity and reconfigurability of the runtime itself. The second contribution, a component-based approach (LooCI), provides modularity both in the runtime and at application development time. DAViM and LooCI focus on different points in the trade-off between fine-grained reconfiguration (DAViM) and clean modularization (LooCI).

Next, Chapter 4 presented a management solution (QARI) for assembly and deployment of distributed software applications that provides autonomy and full multi-application support on top of a modular runtime system. QARI supports a high-level abstraction that allows administrators to easily express the goals for distributed application deployment. QARI autonomously pursues these goals and manages the trade-off between multiple applications. QARI integrates with LooCI because the cleaner modularization of LooCI's component-based approach eases application assembly and promotes component reuse.

The prototype implementations of DAViM, LooCI and QARI, presented in Chapter 5, confirm the feasibility of the approach for resource constrained wireless sensor networks. Two representative application scenarios in transport and logistics illustrated the features of the solutions. Finally, the text reported on the integration of the prototypes of QARI and LooCI in a testbed with state-of-the-art sensor node hardware.

To conclude, the work in this dissertation addresses the problem statement in Section 1.3. This dissertation describes the design and implementation of resource efficient solutions that provide autonomous management of multiple distributed applications on top of flexible reconfiguration support in multi-purpose sensor networks.

6.2 Future work

The management solutions in this dissertation provide autonomous management of the distributed applications on a multi-purpose sensor network. The integration of QARI, the application management solution of Chapter 4, and LooCI, the component-based sensor node runtime of Chapter 3, in a small-scale testbed provides initial confirmation of the practical feasibility of the approach (see Chapter 5). However, challenges remain to extend the approach to full scale multi-purpose sensor networks. These challenges are situated in (1) scalable, resource efficient monitoring and analysis, (2) scalable planning and (3) group communication.

The first challenge is situated in the area of monitoring and analysis. This dissertation did not focus on the monitoring and analysis functionality needed to support the autonomous decisions. Investigation and integration of scalable and resource efficient monitoring techniques for multi-purpose sensor networks is therefore an important area of future work. Improved monitoring can broaden the range of network changes that QARI can handle autonomously. In addition, efficient monitoring is crucial for the scalability of the solutions in this dissertation. Planning and execution are discrete task triggered by changes in the network or the goals, while monitoring, in contrast, is a continuous process.

The second challenge lies in providing scalable planning of the deployment. As highlighted in Section 4.5.1, the planning problem in its general form is NP-complete, and thus inherently not scalable. One path to mitigate this problem is to collaborate with experts on planning algorithms to design specialized algorithms that apply problem specific heuristics to improve the scalability. Another approach is to decentralize the planning and structure the problem hierarchically. In such approach, a top-level planning strategy could resolve the goals to the level of zones and delegate further planning within the zones to another planning node. An interesting route for future work is to investigate whether this zone-level planning problem can be made simple enough to enable in-network planning at this level.

The third challenge involves the integration of middleware for adaptive group communication [132]. Despite the frequent group-based interactions for deployment and event dissemination, the prototypes in this dissertation communicate mostly through unicast. Integration of adaptive group communication in LooCI's networking framework is therefore an interesting path for optimization of the communication overhead for code deployment and event dissemination in LooCI.

Apart from the challenges with respect to the real-world scalability of the solutions, other challenges remain. The reconfiguration in the presented component-based runtime system, LooCI, operates at the granularity of components. As explained in Section 3.5, this is a drawback of this approach when compared to the adaptable virtual machine, DAViM. DAViM provides more fine-grained reconfiguration

through its lightweight application scripts. Policy-based reconfiguration [100] is a complementary technique that augments a component-based approach with similar fine-grained reconfiguration. The policies in such system provide a lightweight, declarative way to express functional and non-functional concerns on top of the base component system. For example, a policy could express that *whenever component A and component B interact, these interactions should be encrypted.* A policy engine intercepts the component interactions and executes the applicable policies. Integration of such policy-based reconfiguration into the solutions in this text is an interesting opportunity for future work. This integration requires an extension of LooCI and support for policy deployment in QARI.

This dissertation mainly draws inspiration from the transport and logistics domain. This domain is a representative example of the application context of multi-purpose sensor networks. Other domains that can benefit from multi-purpose sensor networks include industrial automation and mechatronics, traffic and mobility, and health care. These domains thus provide interesting inspiration for future use cases for the solutions in this dissertation.

6.3 Future outlook

Multi-purpose sensor networks are a promising evolution to increase the return on investment and decrease the operational expenses of wireless sensor network technology. In the long term, this could further evolve towards sensor-networks-as-a-service, similar to infrastructure-as-a-service and platform-as-a-service in cloud computing. In this vision, infrastructure administrators provide sensor networks on which third parties can deploy their own services for a fee.

Security of such sensor network service platforms is challenging. In the short term, secure multi-purpose sensor networks can be built by restricting component deployment to trusted components from trusted third parties. In the long term, scalable resource efficient security enforcement becomes indispensable because such sensor networks monitor privacy sensitive data and incorrect data might have significant real-world impact. For example, an HVAC system in a warehouse might decide to increase heating based on bogus data leading to disruption of the cold chain and loss of expensive goods.

For this vision to become a reality, negotiation of quality levels will be required. This dissertation assumes the quality goals for the applications are known. Either a human administrator or a higher-level management system submits these goals to the management solution in a deployment specification. For the vision of sensor-networks-as-a-service to become feasible, higher-level management systems need to support negotiation of quality levels between the different parties involved. These management systems must use higher-level application requirements, such

as data accuracy, to guide the negotiation of the (deployment) quality levels that they submit to QARI. The list of deployment quality goals that QARI currently supports (coverage, distance, and fraction goals) is a non-exhaustive list. Further investigation needs to assess which deployment quality goals are required to support all data quality requirements. This is especially challenging when actuators are added in the more general case of Wireless Sensor and Actuator Networks (WSAN).

This dissertation focused on *managing* the applications on a multi-purpose sensor network. However, to realize the vision of sensor-networks-as-a-service, there also lie challenges in *programming* the applications. Due to the characteristics of wireless sensor networks, programming applications is non-trivial. It requires continuous attention to the memory footprint and processing requirements. In addition, applications need to be built with fault-tolerance and loose-coupling in mind (see Section 3.4.4). Better programming techniques as well as debugging and tool support are needed to decrease the programming complexity. Education also has an important role to play in this context.

In its essence, a wireless sensor network is a distributed system, thus solutions from distributed systems are often applicable. However, the resource constraints force a developer to think about the essential features of these solutions when applying them to sensor networks. In addition, one has to give consideration to the impact of the limited reliability. As a consequence of these challenges, applying techniques from infrastructure-as-a-service and platform-as-a-service approaches in cloud computing to sensor networks is not straightforward. While cloud computing is considered an important evolution for the management of traditional distributed applications in the coming years, the many challenges to adapt these concepts limit the adoption in wireless sensor networks. The work in this dissertation addresses some of these challenges in the context of application management. While sensor-networks-as-a-service may not be ready for prime time, this dissertation presents an important incremental step towards this vision.

Appendix A

Policy specifications: XML schemas

A.1 Composition specification: LooCI SCA extensions

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  xmlns:looci="http://distrinet.cs.kuleuven.be/xmlns/looci/sca"
  xmlns:sca="http://www.osoa.org/xmlns/sca/1.0"
  ecore:documentRoot="LooCISCADocumentRoot"
  targetNamespace="http://distrinet.cs.kuleuven.be/xmlns/looci/sca">

  <import namespace="http://www.osoa.org/xmlns/sca/1.0"/>

  <element name="implementation.looci.osgi" substitutionGroup="sca:implementation"
       type="looci:LooCIOSGiImplementation"/>
  <element name="implementation.looci.contiki"
       substitutionGroup="sca:implementation"
       type="looci:LooCIContikiImplementation"/>
  <element name="implementation.looci.sunspot"
       substitutionGroup="sca:implementation"
       type="looci:LooCISunSPOTImplementation"/>
  <element name="interface.looci" substitutionGroup="sca:interface"
       type="looci:LooCIInterface"/>
  <element name="binding.looci" substitutionGroup="sca:binding"
       type="looci:LooCIBinding"/>
  <complexType name="LooCIOSGiImplementation">
    <complexContent>
      <extension base="looci:LooCIImplementation"/>
    </complexContent>
  </complexType>
  <complexType name="LooCIImplementation">
    <complexContent>
      <extension base="sca:Implementation">
        <attribute name="file" type="xsd:string"/>
```

```
      </extension>
    </complexContent>
  </complexType>
  <complexType name="LooCIContikiImplementation">
    <complexContent>
      <extension base="looci:LooCIImplementation"/>
    </complexContent>
  </complexType>
  <complexType name="LooCISunSPOTImplementation">
    <complexContent>
      <extension base="looci:LooCIImplementation"/>
    </complexContent>
  </complexType>
  <complexType name="LooCIInterface">
    <complexContent>
      <extension base="sca:Interface">
        <attribute ecore:unsettable="false" name="eventtype" type="xsd:byte"/>
      </extension>
    </complexContent>
  </complexType>
  <complexType name="LooCIBinding">
    <complexContent>
      <extension base="sca:Binding"/>
    </complexContent>
  </complexType>
</schema>
```

A.2 Network specification

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  xmlns:qarin="http://distrinet.cs.kuleuven.be/xmlns/qari/network"
       elementFormDefault="qualified"
  targetNamespace="http://distrinet.cs.kuleuven.be/xmlns/qari/network"
       ecore:documentRoot="QARINetworkDocumentRoot">

  <element name="network" type="qarin:Network"/>

  <complexType name="Network" ecore:implements="qarin:Area">
    <sequence>
      <element name="node" type="qarin:Node" maxOccurs="unbounded" minOccurs="0"/>
      <element name="group" type="qarin:Group" minOccurs="0" maxOccurs="unbounded"/>
      <element name="zone" type="qarin:Zone" minOccurs="0" maxOccurs="unbounded"/>
    </sequence>
    <attribute name="targetNamespace" type="anyURI" use="optional"/>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <element name="node" type="qarin:Node"/>

  <complexType name="Node" ecore:implements="qarin:GroupMember qarin:Locatable">
    <attribute name="accepts" type="QName"/>
    <attribute name="range" type="int"/>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <element name="group" type="qarin:Group"/>

  <complexType name="Group" ecore:implements="qarin:GroupMember">
    <sequence>
      <element name="member" minOccurs="1" maxOccurs="unbounded" type="IDREF"
```

```
        ecore:reference="qarin:GroupMember"/>
    </sequence>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <element name="zone" type="qarin:Zone"/>

  <complexType name="Zone" ecore:implements="qarin:Area">
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <complexType name="GroupMember" abstract="true" ecore:interface="true">
    <attribute name="name" type="ID" use="required"/>
  </complexType>

  <complexType name="Area" abstract="true" ecore:interface="true">
    <attribute name="xmin" type="int"/>
    <attribute name="ymin" type="int"/>
    <attribute name="xmax" type="int"/>
    <attribute name="ymax" type="int"/>
    <attribute name="name" type="ID" use="required"/>
  </complexType>

  <complexType name="Locatable" abstract="true" ecore:interface="true">
    <attribute name="x" type="int"/>
    <attribute name="y" type="int"/>
  </complexType>
</schema>
```

A.3 Constraint specification

```
<?xml version="1.0" encoding="UTF-8"?>
<schema targetNamespace="http://distrinet.cs.kuleuven.be/xmlns/qari/constraints"
  elementFormDefault="qualified"
  xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:qaric="http://distrinet.cs.kuleuven.be/xmlns/qari/constraints"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  ecore:documentRoot="QARIConstraintsDocumentRoot">

  <element name="constraints" type="qaric:Constraints"/>

  <complexType name="Constraints">
    <sequence>
      <element name="constraint" type="qaric:Constraint" maxOccurs="unbounded"/>
      <any namespace="##other" processContents="lax" minOccurs="0"
           maxOccurs="unbounded"/>
    </sequence>
    <attribute name="targetNamespace" type="anyURI" use="optional"/>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <element name="constraint" type="qaric:Constraint" abstract="true"/>

  <complexType name="Constraint" abstract="true">
    <attribute name="subject" type="anyURI" />
    <anyAttribute namespace="##any" processContents="lax" />
  </complexType>

  <element name="constraint.maxcomponents" type="qaric:MaximumComponentsConstraint"
    substitutionGroup="qaric:constraint"/>

  <complexType name="MaximumComponentsConstraint">
```

```
    <complexContent>
      <extension base="qaric:Constraint">
        <attribute name="value" type="int"/>
      </extension>
    </complexContent>
  </complexType>
</schema>
```

A.4 Deployment specification

```
<?xml version="1.0" encoding="UTF-8"?>
<schema xmlns="http://www.w3.org/2001/XMLSchema"
  xmlns:ecore="http://www.eclipse.org/emf/2002/Ecore"
  xmlns:qarid="http://distrinet.cs.kuleuven.be/xmlns/qari/deployment"
       elementFormDefault="qualified"
  targetNamespace="http://distrinet.cs.kuleuven.be/xmlns/qari/deployment"
  ecore:documentRoot="QARIDeploymentDocumentRoot">

  <element name="componentdeployment" type="qarid:ComponentDeploymentRequest" />

  <complexType name="ComponentDeploymentRequest">
    <sequence>
      <any maxOccurs="unbounded" minOccurs="0" namespace="##other"
           processContents="lax"/>
      <element name="target" type="qarid:Target" maxOccurs="unbounded"
           minOccurs="1"/>
      <element name="dependency" type="qarid:Dependency" maxOccurs="unbounded"
           minOccurs="0"/>
    </sequence>
    <attribute name="component" type="QName" use="required"/>
    <attribute name="name" type="NCName" use="optional"/>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <simpleType name="listOfQNames">
    <list itemType="QName"/>
  </simpleType>

  <element name="deployment" type="qarid:DeploymentRequest"/>

  <complexType name="DeploymentRequest">
    <sequence>
      <element name="componentdeployment" type="qarid:ComponentDeploymentRequest"
        maxOccurs="unbounded" minOccurs="0"/>
      <any maxOccurs="unbounded" minOccurs="0" namespace="##other"
           processContents="lax"/>
    </sequence>
    <attribute name="name" type="NCName"/>
    <attribute name="composites" type="qarid:listOfQNames"/>
    <attribute name="targetNamespace" type="anyURI" use="optional"/>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <complexType name="Target">
    <sequence>
      <element name="goal" type="qarid:Goal" maxOccurs="unbounded" minOccurs="0"/>
      <any namespace="##other" processContents="lax" minOccurs="0"
           maxOccurs="unbounded"/>
    </sequence>
    <attribute name="name" type="anyURI"/>
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>
```

```
  <complexType name="Goal" abstract="true">
    <anyAttribute namespace="##any" processContents="lax"/>
  </complexType>

  <complexType name="CoverageGoal">
    <complexContent>
      <extension base="qarid:Goal">
        <attribute name="area" type="string"/>
        <attribute name="min" type="int"/>
        <attribute name="pref" type="int"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="DistanceGoal">
    <complexContent>
      <extension base="qarid:Goal">
        <attribute name="nominal" type="int"/>
        <attribute name="max" type="int"/>
        <attribute name="to" type="QName"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="FractionGoal">
    <complexContent>
      <extension base="qarid:Goal">
        <attribute name="min" type="float"/>
        <attribute name="value" type="float"/>
      </extension>
    </complexContent>
  </complexType>

  <complexType name="Dependency">
    <attribute name="element" type="string"/>
    <attribute name="uri" type="anyURI"/>
    <anyAttribute/>
  </complexType>

  <element name="goal.coverage" type="qarid:CoverageGoal"
       substitutionGroup="qarid:goal"/>
  <element name="goal.fraction" type="qarid:FractionGoal"
       substitutionGroup="qarid:goal"/>
  <element name="goal.distance" type="qarid:DistanceGoal"
       substitutionGroup="qarid:goal"/>
  <element name="goal" type="qarid:Goal" abstract="true"/>
  <element name="target" type="qarid:Target"/>
  <element name="dependency" type="qarid:Dependency"/>
</schema>
```

Appendix B

LooCI/Contiki API

Below is the documentation for the public API of LooCI on Contiki, intended for use by components.

B.1 The peer library

This library abstracts the specific addressing scheme used by LooCI for the node identifiers. The library also makes sure each (potentially large) node identifier is stored in memory only once. The LooCI/Contiki implementation uses the Contiki uIPv6 networking stack and as a consequence node identifiers are IPv6 addresses. Since each IPv6 address is 16 bytes, the use of this library offers a huge advantage in terms of memory consumption.

B.1.1 Define Documentation

#define PEER_ID_ALL 1

The peer id that will be assigned to the 'broadcast peer'.

In the current IPv6 implementation, this is the 'all nodes on this link' multicast address FF02::1

#define PEER_ID_NONE 0

The peer id that will never be assigned to a real peer.

If this peer id is returned, it indicates a failure

B.1.2 Function Documentation

CCIF peer_id_t peer_add (peer_addr_t * *addr*)

Add a peer to the peer list.

Parameters

addr	The address of the new peer

Returns

The id assigned to the peer

CCIF peer_addr_t * peer_get_addr (peer_id_t *id*)

Get the address of a peer.

Parameters

id	The peer id

Returns

The address of the peer. NULL if the id is not known.

Note

The returned pointer is NOT valid across Contiki process waits!!!

CCIF peer_id_t peer_get_id (peer_addr_t * *addr*)

Get the id of a peer.

Parameters

addr	The address of the peer

Returns

The id of the peer. PEER_ID_NONE if the peer isn't in the list yet.

CCIF peer_id_t peer_get_id_or_add (peer_addr_t * *addr*)

Get the id of a peer or add it if the peer is not yet known.

Parameters

addr	The address of the peer

Returns

The id of the peer.

CCIF void peer_remove (peer_id_t *id*)

Remove a peer from the peer list.

Parameters

id	The peer id

Note

All peer_addr_t pointers become invalid by calling this method. They must be rerequested from the peers library with a call to peer_get_addr().

B.2 Event types

B.2.1 Typedef Documentation

typedef u8_t looci_eventtype_t

The LooCI event type.

B.2.2 Function Documentation

CCIF int looci_eventtype_isa (looci_eventtype_t *first*, looci_eventtype_t *second*)

Check the 'is a' relationship between two event types.

Checks whether first 'is a' second.

Parameters

first	The event type to check.
second	The event type to check against.

Returns

Non-zero if the 'is a' relationship holds, zero if it doesn't hold.

B.3 Components

B.3.1 Data Structure Documentation

struct component

A component.

Data Fields

- struct **component** * **next**
- u8_t **id**
- struct **process** * **process**
- **looci_eventtype_t** * **interfaces**
- **looci_eventtype_t** * **receptacles**
- enum **component_state state**

B.3.2 Enumeration Type Documentation

enum component_state

The component states.

Enumerator:

COMPONENT_STATE_NONE

COMPONENT_STATE_ACTIVE
COMPONENT_STATE_DEACTIVATED

B.3.3 Define Documentation

#define COMPONENT(*name, ctype*)

Declare a component.

Parameters

name	The name of the component struct that will be created.
ctype	A string defining the component type.

#define COMPONENT_INTERFACES(*name, ...*)

Declare the interfaces of a component.

Parameters

name	The name of the component.
...	The eventtypes the component publishes.

Note

One of COMPONENT_INTERFACES or COMPONENT_NO_INTERFACES is mandatory! The declaration must be done before the component declaration!

#define COMPONENT_NO_INTERFACES(*name*)

Declare the absence of interfaces on a component.

Parameters

name	The name of the component.

Note

One of COMPONENT_INTERFACES or COMPONENT_NO_INTERFACES is mandatory! The declaration must be done before the component declaration!

#define COMPONENT_NO_RECEPTACLES(*name*)

Declare the absence of receptacles on a component.

Parameters

name	The name of the component.

Note

One of COMPONENT_RECEPTACLES or COMPONENT_NO_RECEPTACLES is mandatory! The declaration must be done before the component declaration!

#define COMPONENT_RECEPTACLES(*name, ...*)

Declare the receptacles of a component.

Parameters

name	The name of the component.
...	The eventtypes the component consumes.

Note

One of COMPONENT_RECEPTACLES or COMPONENT_NO_RECEPTACLES is mandatory! The declaration must be done before the component declaration!

#define COMPONENT_THREAD(*name, ev, data*)

Declare a component thread.

Parameters

name	The component name (declared with COMPONENT())
ev	The Contiki event that caused the component to be scheduled
data	The data passed along with the Contiki event

B.4 Events

B.4.1 Data Structure Documentation

struct looci_event

A LooCI event.

Data Fields

- **looci_eventtype_t type**
- **peer_id_t source_node**
- u8_t **source_cid**
- u8_t **len**
- char **payload** [LOOCI_EVENT_PAYLOAD_MAXLEN]

B.4.2 Define Documentation

#define LOOCI_SUCCESS

Successful.

This value is returned by the LooCI Event Bus if an operation was successful.

#define LOOCI_ERR_EVENTNOTPUBLISHED

Event could not be published.

This value is returned by the LooCI Event Bus if an event could not be published successfully.

#define LOOCI_EVENT_PAYLOAD_MAXLEN 40

The maximum length of a LooCI event payload.

The payload of a LooCI cannot be larger than this constant. If a component tries to publish an event with a larger payload, the passed payload will be truncated!

#define LOOCI_EVENT_RECEIVE(*event*)

Wait until an event is received.

Parameters

event	Pointer to the event after the event is received. This pointer should point to an allocated event structure. The event will be copied into the memory pointed to by the pointer.

#define LOOCI_EVENT_RECEIVE_UNTIL(*event*, *condition*)

Wait until an event is received or condition is true.

Parameters

event	Pointer to the event after the event is received. This pointer should point to an allocated event structure. The event will be copied into the memory pointed to by the pointer.
condition	Alternative condition that will interupt the waiting for an event when the condition becomes true.

#define LOOCI_EVENT_RECEIVED

Condition to test whether an event is received.

Can be used to test whether LOOCI_EVENT_RECEIVE_UNTIL stopped because an event was received.

B.4.3 Function Documentation

CCIF int looci_event_publish (looci_eventtype_t *type*, void * *payload*, u8_t *len*)

Publish a LooCI event.

Parameters

type	The event type.
payload	The event payload.
len	The length of the payload.

Return values

LOOCI_SUCCESS	The event was published successful

LOOCI_ERR_EVENTNOT-PUBLISHED	The event could not be published

Appendix C

LooCI/Contiki reconfiguration and introspection API

This appendix documents the reconfiguration and introspection API for LooCI on Contiki.

C.1 Common definitions

C.1.1 Define Documentation

#define COMPONENT_ID_NONE 0

The null component identifier.

C.2 Deployment

C.2.1 Function Documentation

uint8_t looci_deploy (char * *componentFile*, char * *nodeID*)

Deploy a component to a node.

Parameters

componentFile	The file containing the code for the component.
nodeID	The identifier of the node (IPv6 address).

Returns

The componentID if successfull, COMPONENT_ID_NODE in case of failure.

bool looci_remove (uint8_t *componentID*, char * *nodeID*)

Remove a component.

Parameters

componentID	The identifier of the component to remove.
nodeID	The identifier of the node (IPv6 address) to remove the component from.

Returns

True if the removal was successful, false otherwise.

C.3 Runtime control

C.3.1 Function Documentation

bool looci_activate (uint8_t *componentID*, char * *nodeID*)

Reactivate a component.

Parameters

componentID	The identifier of the component to reactivate.
nodeID	The identifier of the node (IPv6 address) on which to reactivate the component.

Returns

True if the reactivation was successful, false otherwise.

bool looci_deactivate (uint8_t *componentID*, char * *nodeID*)

Deactivate a component.

Parameters

componentID	The identifier of the component to deactivate.
nodeID	The identifier of the node (IPv6 address) on which to deactivate the component.

Returns

True if the deactivation was successful, false otherwise.

bool looci_reset_wirings (uint8_t *componentID*, char * *nodeID*)

Reset the wirings of a component.

Parameters

componentID	The identifier of the component to reset the wiring of.
nodeID	The identifier of the node (IPv6 address) on which to the component resides.

Returns

True if the reset was successful, false otherwise.

bool looci_unwire_from (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, char * *sourceNodeID*, uint8_t *receptacleEvent*, uint8_t *destComponentID*, char * *destNodeID*)

Stop delivering events produced at a specific interface to a specific receptacle.

This doesn't affect the source node.

Parameters

interfaceEvent	The event type at the source.
sourceComponentID	The component identifier at the source.
sourceNodeID	The identifier of the source node (IPv6 address).
receptacleEvent	The event type at the destination.
destComponentID	The component identifier at the destination.
destNodeID	The node identifier of the destination node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_unwire_from_all (uint8_t *interfaceEvent*, uint8_t *receptacleEvent*, uint8_t *destComponentID*, char * *destNodeID*)

Stop delivering certain types of events (regardless of the source node) to a component.

Parameters

interfaceEvent	The event type at the source.
receptacleEvent	The event type at the destination.
destComponentID	The component identifier at the destination.
destNodeID	The node identifier of the destination node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_unwire_local (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, uint8_t *receptacleEvent*, uint8_t *destComponentID*, char * *nodeID*)

Unwire two specific components on the same node.

Parameters

interfaceEvent	The event type at the source.
sourceComponentID	The component identifier at the source.
receptacleEvent	The event type at the destination.
destComponentID	The component identifier at the destination.
nodeID	The node identifier of the node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_unwire_to (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, char * *sourceNodeID*, char * *destNodeID*)

Stop sending events published at a specific interface to a specific node This doesn't affect the destination node.

Parameters

interfaceEvent	The event type at the source.
sourceComponentID	The component identifier at the source.
sourceNodeID	The identifier of the source node (IPv6 address).
destNodeID	The node identifier of the destination node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_unwire_to_all (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, char * *sourceNodeID*)

Stop sending events published at a specific interface to the broadcast address.

Parameters

interfaceEvent	The event type to broadcast.
sourceComponentID	The component identifier of the event source.
sourceNodeID	The identifier of the source node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_wire_from (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, char * *sourceNodeID*, uint8_t *receptacleEvent*, uint8_t *destComponentID*, char * *destNodeID*)

Start delivering events produced at a specific interface to a specific receptacle.

This doesn't affect the source node (see looci_wire_to for that).

Parameters

interfaceEvent	The event type at the source.

sourceComponentID	The component identifier at the source.
sourceNodeID	The identifier of the source node (IPv6 address).
receptacleEvent	The event type at the destination.
destComponentID	The component identifier at the destination.
destNodeID	The node identifier of the destination node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_wire_from_all (uint8_t *interfaceEvent*, uint8_t *receptacleEvent*, uint8_t *destComponentID*, char * *destNodeID*)

Start delivering a certain type of events (regardles of the source node) to a certain receptacle.

Parameters

interfaceEvent	The event type at the source.
receptacleEvent	The event type at the destination.
destComponentID	The component identifier at the destination.
destNodeID	The node identifier of the destination node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_wire_local (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, uint8_t *receptacleEvent*, uint8_t *destComponentID*, char * *nodeID*)

Wire two specific components on the same node.

Parameters

interfaceEvent	The event type at the source.
sourceComponentID	The component identifier at the source.
receptacleEvent	The event type at the destination.
destComponentID	The component identifier at the destination.
nodeID	The node identifier of the node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_wire_to (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, char * *sourceNodeID*, char * *destNodeID*)

Send events published at a specific interface to a specific node.

This doesn't affect the destination node (see looci_wire_from for that).

Parameters

interfaceEvent	The event type at the source.
sourceComponentID	The component identifier at the source.
sourceNodeID	The identifier of the source node (IPv6 address).
destNodeID	The node identifier of the destination node (IPv6 address).

Returns

True if successful, false otherwise.

bool looci_wire_to_all (uint8_t *interfaceEvent*, uint8_t *sourceComponentID*, char * *sourceNodeID*)

Send events published at a specific interface to the broadcast address.

Parameters

interfaceEvent	The event type to broadcast.
sourceComponentID	The component identifier of the event source.
sourceNodeID	The identifier of the source node (IPv6 address).

Returns

True if successful, false otherwise.

C.4 Introspection

C.4.1 Data Structure Documentation

struct looci_incoming_wire

Data type for an incoming wire.

Data Fields

- char **nodeID** [WIRE_NODEID_LENGTH]
- uint8_t **componentID**

struct looci_outgoing_wire

Data type for an outgoing wire.

Data Fields

- char **nodeID** [WIRE_NODEID_LENGTH]

C.4.2 Define Documentation

#define WIRE_NODEID_LENGTH 40

C.4.3 Enumeration Type Documentation

anonymous enum

The component states.

Enumerator:

COMPONENT_STATE_NONE
COMPONENT_STATE_ACTIVE
COMPONENT_STATE_DEACTIVATED

C.4.4 Function Documentation

bool looci_get_component_ids (char * *nodeID*, uint8_t * *buffer*, size_t * *size*)

Get the components on a given node.

Parameters

nodeID	The node identifier (IPv6 address) of the node.
buffer	A buffer where the component ids will be copied to.
size	Pointer to a variable that will contain the number of component ids in the buffer after the call. This variable must be initialized to the maximum number of component ids the buffer can hold.

Returns

'true' if successful, 'false' otherwise

Note

if the return value is false and $*size > 0$, the buffer was too small to hold all component ids. if the return value is false and $*size == 0$, another error occured. if the return value is true and $*size == 0$, there are no components.

bool looci_get_component_ids_by_type (char * *componentType*, char * *nodeID*, uint8_t * *buffer*, size_t * *size*)

Get the components of a given type on a given node.

Parameters

componentType	The type of the components.
nodeID	The node identifier (IPv6 address) of the node.
buffer	A buffer where the component ids will be copied to.
size	Pointer to a variable that will contain the number of component ids in the buffer after the call. This variable must be initialized to the maximum number of component ids the buffer can hold.

Returns

'true' if successful, 'false' otherwise

Note

if the return value is false and ∗size > 0, the buffer was too small to hold all component ids. if the return value is false and ∗size == 0, another error occured. if the return value is true and ∗size == 0, there are no components with the given type.

bool looci_get_componenttype (uint8_t *componentID*, char ∗ *nodeID*, char ∗ *type*, size_t *maxlen*)

Get the type of a given component on a given node.

Parameters

componentId	The component identifier.
nodeID	The node identifier (IPv6 address) of the node.
type	A string buffer where the type will be copied to.
maxlen	The maximum length of a string that the buffer can hold.

Returns

'true' is successful, 'false' otherwise

bool looci_get_incoming_wires (uint8_t *interfaceEvent*, uint8_t *componentID*, char ∗ *nodeID*, struct looci_incoming_wire ∗ *buffer*, size_t ∗ *size*)

Introspect the incoming wires for component.

Parameters

interfaceEvent	The event type of the interface that is wired to a receptacle of the component.
componentID	The component identifier.
nodeID	The identifier of the node (IPv6 address).
buffer	A buffer where the wires will be written to.
size	A pointer to a variable that will hold the number of wires in the buffer after the call. This variable must be initialized to the maximum size of the buffer.

Returns

'true' if succesful, 'false' otherwise

Note

if the return value is false and ∗size > 0, the buffer was too small to hold all wires. if the return value is false and ∗size == 0, another error occured. if the return value is true and ∗size == 0, there are no incoming remote wires for the given component and given interfaceEvent

bool looci_get_interfaces (uint8_t *componentID*, char ∗ *nodeID*, uint8_t ∗ *buffer*, size_t ∗ *size*)

Introspect the interfaces of a component.

Parameters

componentID	The component identifier.
nodeID	The node identifier (IPv6 address) of the node the component is on.
buffer	A buffer where the interfaces will be copied to
size	Pointer to variable that will contain the number of interfaces in the buffer after the call. This variable must be initialized to the maximum number of interfaces the buffer can hold.

Returns

'true' if successful, 'false' otherwise

Note

if the return value is false and ∗size > 0, the buffer was too small to hold all interfaces. if the return value is false and ∗size==0, an other error occured. if the return value is true and ∗size==0, the component has no interfaces

bool looci_get_local_wires (uint8_t *interfaceEvent*, uint8_t *componentID*, char ∗ *nodeID*, uint8_t ∗ *buffer*, size_t ∗ *size*)

Introspect the local wires for a specific interface.

Parameters

interfaceEvent	The event type of the interface.
componentID	The component identifier of the interface.
nodeID	The node identifier (IPv6 address).
buffer	A buffer where the destination component ids will be copied to.

size	Pointer to a variable that will contain the number of component ids in the buffer after the call. This variable must be initialized to the maximum number of component ids the buffer can hold.

Returns

'true' if successful, 'false' otherwise

Note

if the return value is false and ∗size > 0, the buffer was too small to hold all destination component ids. if the return value is false and ∗size == 0, another error occured. if the return value is true and ∗size == 0, there are no local wires for the given interface.

bool looci_get_outgoing_wires (uint8_t *interfaceEvent*, uint8_t *componentID*, char ∗ *nodeID*, struct looci_outgoing_wire ∗ *buffer*, size_t ∗ *size*)

Introspect the outgoing wires for a specific interface.

Parameters

interfaceEvent	The event type of the interface.
componentID	The component identifier of the interface.
nodeID	The identifier of the node (IPv6 address).
buffer	A buffer where the outgoing wires will be written to.
size	A pointer to a variable that will hold the number of wires in the buffer after the call. This variable must be initialized to the maximum size of the buffer.

Returns

'true' if succesful, 'false' otherwise

Note

if the return value is false and ∗size > 0, the buffer was too small to hold all wires. if the return value is false and ∗size == 0, another error occured. if the return value is true and ∗size == 0, there are no outgoing remote wires for the given interface.

bool looci_get_receptacles (uint8_t *componentID*, char * *nodeID*, uint8_t * *buffer*, size_t * *size*)

Introspect the receptacles of a component.

Parameters

componentID	The component identifier.
nodeID	The node identifier (IPv6 address) of the node the component is on.
buffer	A buffer where the receptacles will be copied to
size	Pointer to variable that will contain the number of receptacles in the buffer after the call. This should be initialized to the maximum number of receptacles the buffer can hold.

Returns

'true' if successful, 'false' otherwise

Note

if the return value is false and $*size > 0$, the buffer was too small to hold all receptacles. if the return value is false and $*size == 0$, an other error occured. if the return value is true and $*size == 0$, the component has no receptacles.

uint8_t looci_get_state (uint8_t *componentID*, char * *nodeID*)

Introspect the state of a component.

Parameters

componentID	The component identifier.
nodeID	The node identifier (IPv6 address) of the node the component is on.

Returns

The state of the component. COMPONENT_STATE_NONE if an error of some kind occurred.

C.5 Shell wrapper

```
Usage: looci_shell <command> <command parameters>
Commands
========
Deployment:
-----------
   deploy <file> <node>
   removeComponent <component id> <node>
Runtime control:
----------------
   activate <component id> <node>
   deactivate <component id> <node>
   resetWirings <component id> <node>
   wireLocal <interface event> <source component id> <receptacle event>
        <destination component id> <node>
   unwireLocal <interface event> <source component id> <receptacle event>
        <destination component id> <node>
   wireTo <interface event> <source component id> <source node> <destination node>
   unwireTo <interface event> <source component id> <source node> <destination node>
   wireFrom <interface event> <source component id> <source node> <receptacle
        event> <destination component id> <destination node>
   unwireFrom <interface event> <source component id> <source node> <receptacle
        event> <destination component id> <destination node>
   wireToAll <interface event> <source component id> <source node>
   unwireToAll <interface event> <source component id> <source node>
   wireFromAll <interface event> <receptacle event> <destination component id>
        <destination node>
   unwireFromAll <interface event> <receptacle event> <destination component id>
        <destination node>
Introspection:
--------------
   getComponentIDs <component type> <node>
   getComponentIDs <node>
   getComponentType <component id> <node>
   getState <component id> <node>
   getInterfaces <component id> <node>
   getReceptacles <component id> <node>
   getOutgoingRemoteWires <event id> <source component id> <source node id>
   getIncomingRemoteWires <interface event id> <destination component id>
        <destination node id>
   getLocalWires <event id> <source component id> <node>
```

Bibliography

[1] Karl Aberer, Manfred Hauswirth, and Ali Salehi. A middleware for fast and flexible sensor network deployment. In *Very Large Data Bases (VLDB)*, 2006.

[2] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Infrastructure for data processing in large-scale interconnected sensor networks. In *Mobile Data Management (MDM)*, 2007.

[3] Karl Aberer, Manfred Hauswirth, and Ali Salehi. Zero-programming sensor network deployment. In *Next Generation Service Platforms for Future Mobile Systems (SPMS)*, 2007.

[4] William Arnold, Tamar Eilam, Michael Kalantar, Alexander Konstantinou, and Alexander Totok. Pattern based soa deployment. In Bernd Krämer, Kwei-Jay Lin, and Priya Narasimhan, editors, *Service-Oriented Computing - ICSOC 2007*, volume 4749 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2007.

[5] William Arnold, Tamar Eilam, Michael Kalantar, Alexander Konstantinou, and Alexander Totok. Automatic realization of soa deployment patterns in distributed environments. In Athman Bouguettaya, Ingolf Krueger, and Tiziana Margaria, editors, *Service-Oriented Computing - ICSOC 2008*, volume 5364 of *Lecture Notes in Computer Science*, pages 162–179. Springer Berlin / Heidelberg, 2008.

[6] Naveed Arshad, Dennis Heimbigner, and Alexander L. Wolf. Deployment and dynamic reconfiguration planning for distributed software systems. *Software Quality Control*, 15(3):265–281, 2007.

[7] Atmel Corporation. The RZRAVEN 2.4 ghz evaluation and starter kit. `http://www.atmel.com/dyn/products/tools_card.asp?tool_id=4291`, April 2010.

[8] Edgardo Avilés-López and J Antonio García-Macías. TinySOA: a service-oriented architecture for wireless sensor networks. *Service Oriented Computing and Applications*, 3:99–108, 2009. 10.1007/s11761-009-0043-x.

[9] Asad Awan, Suresh Jagannathan, and Ananth Grama. Macroprogramming heterogeneous sensor networks using cosmos. In *Proceedings of EuroSys 2007*, March 2007.

[10] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, Berkeley, CA, USA, 2005. USENIX Association.

[11] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *ACM Conference on Embedded Systems Software (EMSOFT)*, October 2006.

[12] D Balasubramaniam, A Dearle, and R Morrison. A composition-based approach to the construction and dynamic reconfiguration of wireless sensor network applications. In *7th International Symposium on Software Composition (SC 2008), Budapest, Hungary*, pages 206–214. Springer Verlag, 2008.

[13] C. Bidan, V. Issarny, T. Saridakis, and A. Zarras. A dynamic reconfiguration service for corba. In *Configurable Distributed Systems, 1998. Proceedings. Fourth International Conference on*, pages 35–42, May 1998.

[14] Urs Bischoff and Gerd Kortuem. Rulecaster: A macroprogramming system for sensor networks. In *Proceedings of the OOPSLA '06 Workshop on Building Software for Sensor Networks*, October 2006.

[15] Jim Blythe, Ewa Deelman, A Gil, Carl Kesselman, Amit Agarwal, Gaurang Mehta, and Karan Vahi. The role of planning in grid computing. In *13th International Conference on Automated Planning and Scheduling (ICAPS)*, 2003.

[16] S. Bouchenak, N. De Palma, D. Hagimont, and C. Taton. Autonomic management of clustered applications. In *Cluster Computing, 2006 IEEE International Conference on*, pages 1–11, September 2006.

[17] Eric Bouillet, Mark Feblowitz, Hanhua Feng, Anand Ranganathan, Anton Riabov, Octavian Udrea, and Zhen Liu. Mario: middleware for assembly and deployment of multi-platform flow-based applications. In *Middleware '09: Proceedings of the 10th ACM/IFIP/USENIX International Conference on Middleware*, pages 1–7, New York, NY, USA, 2009. Springer-Verlag New York, Inc.

[18] R.A. Brooks. The intelligent room project. In *Cognitive Technology, 1997. 'Humanizing the Information Age'. Proceedings., Second International Conference on*, pages 271–278, August 1997.

[19] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The fractal component model and its support in java. *Software-Practice and Experience*, 36:1257–1284, 2006.

[20] Eddy Caron, Pushpinder Kaur Chouhan, and Holly Dail. GoDIET: a deployment tool for distributed middleware on grid'5000. Technical Report 5886, INRIA, April 2006.

[21] Eddy Caron and Holly Dail. GoDIET: a tool for managing distributed hierarchies of DIET agents and servers. Technical Report 5520, INRIA, March 2005.

[22] Hakima Chaouchi, editor. *The Internet of Things: Connecting Objects*. Wiley-ISTE, May 2010.

[23] Chee-Yee Chong and S.P. Kumar. Sensor networks: evolution, opportunities, and challenges. *Proceedings of the IEEE*, 91(8):1247–1256, August 2003.

[24] Pietro Ciciriello, Luca Mottola, and Gian Pietro Picco. Efficient routing from multiple sources to multiple sinks in wireless sensor networks. In *Proceedings of the 4th European Conference on Wireless Sensor Networks (EWSN07)*, January 2007.

[25] Lynn Conway, Victor R. Lesser, and Daniel G. Corkill. The distributed vehicle monitoring testbed: A tool for investigating distributed problem solving networks. *AI Magazine*, 4(3):15–33, 1983.

[26] Massimo Coppola, Nicola Tonellotto, Marco Danelutto, Corrado Zoccolo, Sébastien Lacour, Christian Pérez, and Thierry Priol. Towards a common deployment model for grid systems. In Sergei Gorlatch and Marco Danelutto, editors, *Integrated Research in GRID Computing*, pages 15–30. Springer US, 2007.

[27] Sentilla Corporation. Sentilla website. `http://www.sentilla.com/`, August 2008.

[28] Paolo Costa, Geoff Coulson, Cecilia Mascolo, Luca Mottola, Gian Pietro Picco, and Stefanos Zachariadis. Reconfigurable component-based middleware for networked embedded systems. *International Journal of Wireless Information Networks*, 2006.

[29] Geoff Coulson, Gordon Blair, Paul Grace, Francois Taiani, Ackbar Joolia, Kevin Lee, Jo Ueyama, and Thirunavukkarasu Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1–42, 2008.

[30] Crossbow Technology, Inc. Imote2 hardware reference manual, revision A. Available online: `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/6-user-manuals.html?download=56%3Aimote2-hardware-reference-manual`, September 2007.

[31] David Culler, Prabal Dutta, Cheng Tien Ee, Rodrigo Fonseca, Jonathan Hui, Philip Levis, Joseph Polastre, Scott Shenker, Ion Stoica, Gilman Tolle, and Jerry Zhao. Towards a sensor network architecture: lowering the waistline. In *Proceedings of the 10th conference on Hot Topics in Operating Systems - Volume 10*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[32] Eric M. Dashofy, André van der Hoek, and Richard N. Taylor. Towards architecture-based self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 21–26, New York, NY, USA, 2002. ACM.

[33] Alan Dearle, Dharini Balasubramaniam, Jonathan Lewis, and Ron Morrison. A component-based model and language for wireless sensor network applications. *Computer Software and Applications Conference, Annual International*, 0:1303–1308, 2008.

[34] Alan Dearle, Graham Kirby, Andrew McCarthy, and Juan Carlos Diaz y Carballo. A flexible and secure deployment framework for distributed applications. In *Component Deployment*, volume 3083/2004 of *Lecture Notes in Computer Science*, pages 219–233. Springer Berlin / Heidelberg, May 2004.

[35] Ewa Deelman, James Blythe, Yolanda Gil, Carl Kesselman, Gaurang Mehta, Karan Vahi, Kent Blackburn, Albert Lazzarini, Adam Arbree, Richard Cavanaugh, and Scott Koranda. Mapping abstract complex workflows onto grid environments. *Journal of Grid Computing*, 1:25–39, 2003.

[36] Thomas Delaet. *Improving Software and Data Management in Contemporary IT Systems (Het verbeteren van software- en gegevens-beheer in hedendaagse IT systemen)*. PhD thesis, IWT, December 2009.

[37] Thomas Delaet and Wouter Joosen. Podim: a language for high-level configuration management. In *Proceedings of the 21st Large Installation System Administration Conference (USENIX LISA '07)*, pages 261–273. Usenix Association, November 2007.

[38] Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. A survey of system configuration tools. In *Proceedings of the 24th Large Installation System Administration Conference (LISA'10)*, November 2010.

[39] Adam Dunkels, Niclas Finne, Joakim Eriksson, and Thiemo Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings*

of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006), Boulder, Colorado, USA, November 2006.

[40] Adam Dunkels, Richard Gold, Sergio Angel Marti, Arnold Pears, and Mats Uddenfeldt. Janus: An architecture for flexible access to sensor networks. In *Proceedings of First International ACM Workshop on Dynamic Interconnection of Networks (DIN'05)*, Cologne, Germany, 2005.

[41] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN'04)*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[42] Adam Dunkels, Fredrik Osterlind, Nicolas Tsiftes, and Zhitao He. Software-based on-line energy estimation for sensor nodes. In *Proceedings of the 4th workshop on Embedded networked sensors*, EmNets '07, pages 28–32, New York, NY, USA, 2007. ACM.

[43] Christos Efstratiou, Ilias Leontiadis, Cecilia Mascolo, and Jon Crowcroft. Demo abstract: A shared sensor network infrastructure. In *In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (Sensys 2010)*, November 2010.

[44] Matthias Ehrgott and Xavier Gandibleux. A survey and annotated bibliography of multiobjective combinatorial optimization. *OR Spectrum*, 22:425–460, 2000.

[45] T. Eilam, M. H. Kalantar, A. V. Konstantinou, and G. Pacifici. Reducing the complexity of application deployment in large data centers. In *Integrated Network Management, 2005. IM 2005. 2005 9th IFIP/IEEE International Symposium on*, pages 221–234, 2005.

[46] T. Eilam, M.H. Kalantar, A.V. Konstantinou, G. Pacifici, J. Pershing, and A. Agrawal. Managing the configuration complexity of distributed applications in internet data centers. *Communications Magazine, IEEE*, 44(3):166–177, March 2006.

[47] Kaoutar El Maghraoui, Alok Meghranjani, Tamar Eilam, Michael Kalantar, and Alexander V. Konstantinou. Model driven provisioning: bridging the gap between declarative object models and procedural provisioning tools. In *Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, Middleware '06, pages 404–423, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[48] European Commission, Taxation and Customs Union. Authorized Economic Operator (AEO), `http://ec.europa.eu/taxation_customs/customs/policy_issues/customs_security/aeo/index_en.htm`, May 2011.

[49] Areski Flissi, Jérémy Dubus, Nicolas Dolet, and Philippe Merle. Deploying on the grid with DeployWare. In *Proceedings of the 8th International Symposium on Cluster Computing and the Grid (CCGRID'08)*, pages 177–184, Lyon, France, May 2008. IEEE.

[50] David Garlan and Bradley Schmerl. Model-based adaptation for self-healing systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 27–32, New York, NY, USA, 2002. ACM.

[51] David Gay, Philip Levis, Robert V. von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, May 2003. ACM Press.

[52] Ioannis Georgiadis, Jeff Magee, and Jeff Kramer. Self-organising software architectures for distributed systems. In *Proceedings of the first workshop on Self-healing systems*, WOSS '02, pages 33–38, New York, NY, USA, 2002. ACM.

[53] Omprakash Gnawali, Ben Greenstein, Ki-Young Jang, August Joki, Jeongyeup Paek, Marcos Vieira, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *Proceedings of the ACM Conference on Embedded Networked Sensor Systems (Sensys)*, Boulder, Colorado, November 2006.

[54] Patrick Goldsack, Julio Guijarro, Steve Loughran, Alistair Coles, Andrew Farrell, Antonio Lain, Paul Murray, and Peter Toft. The smartfrog configuration management framework. *SIGOPS Oper. Syst. Rev.*, 43(1):16–25, 2009.

[55] Paul Grace, Geoff Coulson, Gordon Blair, Barry Porter, and Danny Hughes. Dynamic reconfiguration in sensor middleware. In *Proceedings of the First International Workshop on Middleware for Sensor Networks (MidSens)*, New York, NY, USA, November 2006. ACM Press.

[56] Ramakrishna Gummadi, Omprakash Gnawali, and Ramesh Govindan. Macro-programming wireless sensor networks using kairos. In *DCOSS*, pages 126–140, 2005.

[57] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM Press.

[58] Chih-Chieh Han, Ram Kumar, Roy Shea, and Mani Srivastava. Sensor network software update management: a survey. *International Journal of Network Management*, 15(4):283–294, 2005.

[59] Jan-Hinrich Hauer, Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. A component framework for content-based publish/subscribe in sensor networks. In *Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN)*. Springer, January 2008.

[60] Wendi B. Heinzelman, Amy L. Murphy, Hervaldo S. Carvalho, and Mark A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1):6–14, 2004.

[61] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *ACM SIGPLAN Notices*, 35(11):93–104, 2000.

[62] Jamie Hillman and Ian Warren. An open framework for dynamic reconfiguration. In *Proceedings of the 26th International Conference on Software Engineering*, ICSE '04, pages 594–603, Washington, DC, USA, 2004. IEEE Computer Society.

[63] Markus C. Huebscher and Julie A. McCann. A survey of autonomic computing—degrees, models, and applications. *ACM Comput. Surv.*, 40:7:1–7:28, August 2008.

[64] Daniel Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Pedro Javier del Cid Garcia, Sam Michiels, Christophe Huygens, Wouter Joosen, and Jo Ueyama. Building wireless sensor network applications with LooCI. *International Journal of Mobile Computing and Multimedia Communications*, 2(4):38–64, October 2010.

[65] Danny Hughes, Jo Ueyama, Eduardo Mendiondo, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens, Wouter Joosen, Ka Man, and Sheng-Uei Guan. A middleware platform to support river monitoring using wireless sensor networks. *Journal of the Brazilian Computer Society*, 17(2):85–102, 2011.

[66] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 81–94, New York, NY, USA, 2004. ACM Press.

[67] Jonathan W. Hui and David E. Culler. Extending ip to low-power, wireless personal area networks. *Internet Computing, IEEE*, 12(4):37–45, July-Aug. 2008.

[68] Christophe Huygens and Wouter Joosen. Federated and shared use of sensor networks through security middleware. In *Proceedings of the Sixth International Conference on Information Technology: New Generations (ITNG'09)*, pages 1005–1011. IEEE, 2009.

[69] IBBT AdMid project. Adaptive Middleware for Logistics, `http://www.ibbt.be/en/projects/overview-projects/p/detail/admid-2`, September 2011.

[70] IBBT Multitrans project. Multimodal transport, `http://www.ibbt.be/en/projects/overview-projects/p/detail/multitrans`, September 2011.

[71] IBM. An architectural blueprint for autonomic computing. White Paper, June 2006.

[72] IEEE. IEEE Std. 802.15.4, Specific requirements Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low Rate Wireless Personal Area Networks (LR-WPANs), December 2003.

[73] IETF. Constrained RESTful environments, description of working group. `http://datatracker.ietf.org/wg/core/charter/`, March 2011.

[74] IETF. Routing over low power and lossy networks, description of working group. `https://datatracker.ietf.org/wg/roll/charter/`, July 2011.

[75] Nico Janssens. *Dynamic software reconfiguration in programmable networks.* PhD thesis, Department of Computer Science, K.U.Leuven, Leuven, Belgium, December 2006.

[76] Nico Janssens, Wouter Joosen, and Pierre Verbaeten. Necoman: middleware for safe distributed-service adaptation in programmable networks. *IEEE Distributed Systems Online*, 6(7):1–11, July 2005.

[77] JBoss Community. JBoss Drools. `http://www.jboss.org/drools`, December 2010.

[78] JBoss Community. JBoss drools planner. `http://www.jboss.org/drools/drools-planner`, December 2010.

[79] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for "smart dust". In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 271–278, New York, NY, USA, 1999. ACM.

[80] J.O. Kephart and W.E. Walsh. An artificial intelligence perspective on autonomic computing policies. In *Policies for Distributed Systems and Networks, 2004. POLICY 2004. Proceedings. Fifth IEEE International Workshop on*, pages 3–12, June 2004.

[81] T. Kichkaylo, A. Ivan, and V. Karamcheti. Constrained component deployment in wide-area networks using ai planning techniques. In *Parallel and Distributed Processing Symposium, 2003. Proceedings. International*, page 10 pp., April 2003.

[82] T. Kichkaylo and V. Karamcheti. Optimal resource-aware deployment planning for component-based distributed applications. In *High performance Distributed Computing, 2004. Proceedings. 13th IEEE International Symposium on*, pages 150–159, June 2004.

[83] Tomasz Kobialka, Rajkumar Buyya, Christopher Leckie, and Ramamohanarao Kotagiri. A sensor web middleware with stateful services for heterogeneous sensor networks. In *Intelligent Sensors, Sensor Networks and Information, 2007. ISSNIP 2007. 3rd International Conference on*, pages 491–496, December 2007.

[84] J. Kramer. Configuration programming-a framework for the development of distributable systems. In *CompEuro '90. Proceedings of the 1990 IEEE International Conference on Computer Systems and Software Engineering*, pages 374 –384, May 1990.

[85] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on Software Engineering*, 16:1293–1306, 1990.

[86] Santosh Kumar, Ten H. Lai, and József Balogh. On k-coverage in a mostly sleeping sensor network. *Wireless Networks*, 14(3):277–294, 2008.

[87] Richard T. Lacoss. Distributed mixed sensor aircraft tracking. In *American Control Conference*, pages 1827 –1830, june 1987.

[88] Koen Langendoen and Niels Reijers. Distributed localization in wireless sensor networks: a quantitative comparison. *Comput. Networks*, 43(4):499–518, 2003.

[89] E.A. Lee. Cyber physical systems: Design challenges. In *Object Oriented Real-Time Distributed Computing (ISORC), 2008 11th IEEE International Symposium on*, pages 363 –369, May 2008.

[90] Philip Levis and David Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM Press.

[91] Philip Levis, David Gay, and David Culler. Active sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, May 2005.

[92] Philip Levis, Samuel Madden, David Gay, Joseph Polastre, Robert Szewczyk, Alec Woo, Eric A. Brewer, and David E. Culler. The emergence of networking abstractions and techniques in tinyos. In *Proc. 1st Symposium on Networked Systems Design and Implementation (NSDI '04)*, pages 1–14, March 2004.

[93] Philip Levis, Neil Patel, David E. Culler, and Scott Shenker. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the First Symposium on Networked Systems Design and Implementation (NSDI 2004)*, pages 15–28, San Francisco, CA, USA, April 2004. USENIX Association.

[94] Shuoqi Li, Sang Hyuk Son, and John A. Stankovic. Event detection services using data service middleware in distributed sensor networks. In *IPSN*, pages 502–517, 2003.

[95] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, New York, NY, USA, 2005. ACM Press.

[96] Jie Liu and Feng Zhao. Towards semantic services for sensor-rich information systems. In *2nd International Conference on Broadband Networks*, pages 44–51, October 2005.

[97] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst.*, 30(1):122–173, 2005.

[98] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In *Proceedings of the Third European Workshop on Wireless Sensor Networks (EWSN 2006)*, pages 212–227, February 2006.

[99] Pedro José Marrón, Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, pages 278–289, January 2005.

[100] Nelson Matthys, Daniel Hughes, Sam Michiels, Christophe Huygens, and Wouter Joosen. Fine-grained tailoring of component behaviour for embedded systems. In *The 7th IFIP Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS), LNCS 5860*, pages 156–167. Springer-Verlag, November 2009.

[101] A.J. McCarthy, A. Dearle, and G.N.C. Kirby. Applying constraint solving to the management of distributed applications. Technical report, University of St Andrews, 2008.

[102] S. Meguerdichian, F. Koushanfar, M. Potkonjak, and M.B. Srivastava. Coverage problems in wireless ad-hoc sensor networks. In *INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE*, volume 3, pages 1380–1387 vol.3, 2001.

[103] MEMSIC. TelosB mote platform datasheet. Available online: `http://www.memsic.com/support/documentation/wireless-sensor-networks/category/7-datasheets.html?download=152%3Atelosb&start=20`, June 2011.

[104] Luca Mottola and Gian Pietro Picco. Programming wireless sensor networks with logical neighborhoods. In *InterSense '06: Proceedings of the first international conference on Integrated internet ad hoc and sensor networks*, page 8, New York, NY, USA, 2006. ACM Press.

[105] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. Fine-grained software reconfiguration for wireless sensor networks. In *Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN)*, 2008.

[106] René Müller and Gustavo Alonso. Efficient sharing of sensor networks. In *Proceedings of the 3rd IEEE International Conference on Mobile Ad-hoc and Sensor Systems 2006*, October 2006.

[107] René Müller, Gustavo Alonso, and Donald Kossmann. Swissqm: Next generation data processing in sensor networks. In *Proceedings of the 3rd Biennial Conference on Innovative Data Systems Research*, January 2007.

[108] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. In *Proceedings of EuroSys 2007*, March 2007.

[109] Ryan Newton and Matt Welsh. Region streams: functional macroprogramming for sensor networks. In *DMSN '04: Proceeedings of the 1st international workshop on Data management for sensor networks*, pages 78–87, New York, NY, USA, 2004. ACM Press.

[110] Open Service Oriented Architectures. SCA service component architecture, assembly model specification. SCA Version 1.00, March 15 2007.

[111] Oracle. Oracle sensor edge server. Technical report, Oracle, February 2006.

[112] Oracle Labs. Sun SPOT world. `http://www.sunspotworld.com/`, June 2011.

[113] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *Software Engineering, 1998. Proceedings of the 1998 International Conference on*, pages 177–186, April 1998.

[114] OSGi Alliance. Benefits of using OSGi. `http://www.osgi.org/About/WhyOSGi`, June 2011.

[115] Ibrahim Osman and Gilbert Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:511–623, 1996. 10.1007/BF02125421.

[116] Barry Porter and Geoff Coulson. Lorien: a pure dynamic component-based operating system for wireless sensor networks. In *Proceedings of the 4th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, MidSens '09, pages 7–12, New York, NY, USA, 2009. ACM.

[117] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 253–266, New York, NY, USA, 2008. ACM.

[118] Abdelmounaam Rezgui and Mohamed Eltoweissy. Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead. *Comput. Commun.*, 30(13):2627–2648, 2007.

[119] Kay Römer, Christian Frank, Pedro José Marrón, and Christian Becker. Generic role assignment for wireless sensor networks. In *Proceedings of the 11th ACM SIGOPS European Workshop*, pages 7–12, September 2004.

[120] Sean Rooney, Daniel Bauer, and Paolo Scotton. Edge server software architecture for sensor applications. In *SAINT '05: Proceedings of the The 2005 Symposium on Applications and the Internet (SAINT'05)*, pages 64–71, Washington, DC, USA, 2005. IEEE Computer Society.

[121] Matthias Rothensee. User acceptance of the intelligent fridge: Empirical results from a simulation. In Christian Floerkemeier, Marc Langheinrich, Elgar Fleisch, Friedemann Mattern, and Sanjay Sarma, editors, *The Internet of Things*, volume 4952 of *Lecture Notes in Computer Science*, pages 123–139. Springer Berlin / Heidelberg, 2008.

[122] Srikanth Sastry, Tsvetomira Radeva, Jianer Chen, and Jennifer L. Welch. Reliable networks with unreliable sensors. In *Distributed Computing and Networking*, volume 6522/2011 of *Lecture Notes in Computer Science*, pages 281–292. Springer, 2011.

[123] Avrora sensor simulator. The avr simulation and analysis framework. `http://compilers.cs.ucla.edu/avrora/`, 2006.

[124] Chavalit Srisathapornphat, Chaiporn Jaikaeo, and Chien-Chung Shen. Sensor information networking architecture. In *2000 International Workshop on Parallel Processing (ICPP 2000)*, Toronto, Canada, August 2000.

[125] STADiUM project. Software Technologies For Adaptable Distributed Middleware, `http://distrinet.cs.kuleuven.be/projects/stadium/`, September 2011.

[126] Thanos Stathopoulos, Tyler McHenry, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical Report 30, Center for Embedded Networked Sensing (CENS), UCLA, 2003.

[127] Jan Steffan, Ludger Fiege, Mariano Cilia, and Alejandro Buchmann. Towards Multi-Purpose wireless sensor networks. In *International Conference on Sensor Networks (SENET'05)*, Montreal, Canada, August 2005. IEEE.

[128] Jun-Zhao Sun. Dissemination protocols for reprogramming wireless sensor networks: A literature survey. In *Fourth International Conference on Sensor Technologies and Applications (SENSORCOMM)*, pages 151 –156, jul. 2010.

[129] Clemens Szyperski. *Component Software - Beyond Object-Oriented Programming*. Addison-Wesley / ACM Press, second edition, 2002. With Dominik Gruntz and Stephan Murer.

[130] Amirhosein Taherkordi, Frederic Loiret, Romain Rouvoy, Azadeh Abdolrazaghi, Quan Le-Trung, and Frank Eliassen. Programming sensor networks using remora component model. In *6th IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS)*, June 2010.

[131] Klaas Thoelen, Danny Hughes, Nelson Matthys, Wouter Horré, Lei Fang, Sheng-Uei Guan, Christophe Huygens, and Wouter Joosen. Supporting reconfiguration and re-use through self-describing component interfaces. In *Proceedings of the fifth International workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens'10)*, 2010.

[132] Klaas Thoelen, Sam Michiels, and Wouter Joosen. Middleware for adaptive group communication in wireless sensor networks. In *Proceedings of the 2nd International ICST conference on Sensor Systems and Software*. ICST, December 2010.

[133] Peter Gober Thomas Luckenbach and Stefan Arbanowski. Tinyrest - a protocol for integrating sensor networks into the internet. In *Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, June 2005.

[134] B. L. Titzer, D. K. Lee, and J. Palsberg. Avrora: scalable sensor network simulation with precise timing. In *Proceedings of Fourth International Symposium on Information Processing in Sensor Networks (IPSN 2005)*, pages 477–482, Los Angeles, CA, USA, April 2005.

[135] Sana Ullah, Henry Higgins, Bart Braem, Benoit Latre, Chris Blondia, Ingrid Moerman, Shahnaz Saleem, Ziaur Rahman, and Kyung Kwak. A comprehensive survey of wireless body area networks. *Journal of Medical Systems*, pages 1–30, 2010.

[136] U.S. Department of Homeland Security, Customs and Border Protection. Container security initiative, 2006-2011 strategic plan.

[137] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE Transactions on Software Engineering*, 33:856–868, 2007.

[138] Giacomino Veltri, Qingfeng Huang, Gang Qu, and Miodrag Potkonjak. Minimal and maximal exposure path algorithms for wireless embedded sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 40–50, New York, NY, USA, 2003. ACM.

[139] Miao-Miao Wang, Jian-Nong Cao, Jing Li, and Sajal K. Das. Middleware for wireless sensor networks: A survey. *Journal of Computer Science and Technology*, 23(3):305–326, May 2008.

[140] Xiaorui Wang, Guoliang Xing, Yuanfang Zhang, Chenyang Lu, Robert Pless, and Christopher Gill. Integrated coverage and connectivity configuration in wireless sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 28–39, New York, NY, USA, 2003. ACM.

[141] Ian Warren and Ian Sommerville. Dynamic configuration abstraction. In Wilhelm Schäfer and Pere Botella, editors, *Software Engineering – ESEC '95*, volume 989 of *Lecture Notes in Computer Science*, pages 173–190. Springer Berlin / Heidelberg, 1995.

[142] M. Weiser. Hot topics-ubiquitous computing. *Computer*, 26(10):71–72, oct 1993.

[143] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh. Deploying a wireless sensor network on an active volcano. *Internet Computing, IEEE*, 10(2):18–25, March-April 2006.

[144] A. Wheeler. Commercial applications of wireless sensor networks using zigbee. *Communications Magazine, IEEE*, 45(4):70–77, April 2007.

[145] Qing Wu and Zhaohui Wu. Adaptive component management service in scudware middleware for smart vehicle space. In Boualem Benatallah, Fabio Casati, and Paolo Traverso, editors, *Service-Oriented Computing - ICSOC 2005*, volume 3826 of *Lecture Notes in Computer Science*, pages 310–323. Springer Berlin / Heidelberg, 2005.

[146] Yong Yao and Johannes Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 31(3):9–18, 2002.

[147] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. Wireless sensor network survey. *Computer Networks*, 52(12):2292–2330, 2008.

List of scientific publications

Articles in international reviewed journals

Wouter Horré, Danny Hughes, Sam Michiels, Wouter Joosen, Advanced sensor network software deployment using application-level quality goals. *Journal of Software*, Vol. 6, No. 4, Pages: 528–535, April 2011

Danny Hughes, Jo Ueyama, Eduardo Mendiondo, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens, Wouter Joosen, Ka Man, Sheng-Uei Guan, A middleware platform to support river monitoring using wireless sensor networks. *Journal of the Brazilian Computer Society*, Volume 17, Issue 2 (2011), Page 85–102

Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Pedro Javier del Cid Garcia, Sam Michiels, Christophe Huygens, Wouter Joosen, Jo Ueyama, Building wireless sensor network applications with LooCI. *International Journal of Mobile Computing and Multimedia Communications*, volume 2, issue 4, pages 38–64, October 2010

Danny Hughes, Kevin Lee, Wouter Horré, Sam Michiels, Ka Lok Man, Wouter Joosen, A graph based approach to supporting software reconfiguration in distributed sensor network applications. *Journal of Internet Technology*, pages 561–571, July 2010

Wouter Horré, Sam Michiels, Wouter Joosen and Pierre Verbaeten, DAVIM: Adaptable Middleware for Sensor Networks, *IEEE Distributed Systems Online*, vol. 9, no. 1, 2008, art. no. 0801-o1001

Contributions at international conferences, published in proceedings

Klaas Thoelen, Nelson Matthys, Wouter Horré, Christophe Huygens, Wouter Joosen, Daniel Hughes, Lei Fang, Sheng-Uei Guan, Supporting reconfiguration and re-use through self-describing component interfaces, *MidSens '10: Proceedings of the 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks*, pages 29–34, Bangalore, 30 November 2010

Jo Ueyama, Daniel Hughes, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens, Wouter Joosen, Applying a multi-paradigm approach to implementing wireless sensor network based river monitoring. In *Proceedings of the IEEE International Workshop on Information Security and Applications*, Qinhuangdao, China, 22-25 October 2010

Jo Ueyama, Daniel Hughes, Nelson Matthys, Wouter Horré, Sam Michiels, Christophe Huygens, Wouter Joosen, An event-based component model for sensor networks: a case study for river monitoring. In *Proceedings of the tools session of the Brazilian symposium on computer networks and distributed systems (SBRC2010)*, Gramado, Brazil, 24-28 May 2010

Wouter Horré, Danny Hughes, Sam Michiels, and Wouter Joosen, QARI: quality aware software deployment for wireless sensor networks. In *Proceedings of the 7th international conference on Information Technology: New Generations (ITNG 2010)*, Las Vegas, Nevada, USA, Apr. 2010.

Wouter Horré, Kevin Lee, Danny Hughes, Sam Michiels, and Wouter Joosen, A graph based approach to supporting reconfiguration in wireless sensor networks. In *Proceedings of the 1st Workshop on Applications of Graph Theory in Wireless Ad hoc Networks and Sensor Networks*, Chennai (Madras), India, Dec. 2009.

Danny Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Javier Del Cid, Sam Michiels, Christophe Huygens, and Wouter Joosen, LooCI: a loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of 7th international conference on advances in Mobile Computing and Multimedia (MoMM09)*, Kuala Lumpur, Malaysia, Dec. 2009.

Wouter Horré, Sam Michiels, Nelson Matthys, Wouter Joosen and Pierre Verbaeten, On the integration of sensor networks and general purpose IT

infrastructure. In *Proceedings of the second international workshop on Middleware for Sensor networks*, November 30, 2007, Newport Beach, CA, USA. ACM International Conference Proceeding Series; ACM, New York, NY, USA. ISBN: 978-1-59593-6

Sam Michiels, Wouter Horré, Wouter Joosen and Pierre Verbaeten, DAViM: a dynamically adaptable virtual machine for sensor networks. In *Proceedings of the international workshop on Middleware for Sensor networks*, November 28, 2006, Melbourne, Australia. ACM International Conference Proceeding Series; Vol. 218. ACM, New York, NY, USA. ISBN: 1-59593-424-3

Technical reports

Daniel Hughes, Klaas Thoelen, Wouter Horré, Nelson Matthys, Pedro Javier del Cid Garcia, Sam Michiels, Christophe Huygens, Wouter Joosen, LooCI: A loosely-coupled component infrastructure for networked embedded systems. *Technical Report (CW Reports), volume CW564*, Department of Computer Science, K.U.Leuven, September 2009

Wouter Horré, Nelson Matthys, Sam Michiels, Wouter Joosen and Pierre Verbaeten, A survey of middleware for wireless sensor networks. *Technical Report (CW Reports), volume CW498*, Department of Computer Science, K.U.Leuven, August 2007

Arenberg Doctoral School of Science, Engineering & Technology
Faculty of Engineering
Department of Computer Science
Research group DistriNet
Celestijnenlaan 200A
B-3001 Leuven