KU LEUVEN

ARENBERG DOCTORAL SCHOOL
Faculty of Engineering Science

An Application Platform for Multi-purpose Sensor Systems

Klaas Thoelen

Supervisor:
Prof. dr. D. Hughes
Prof. dr. ir. W. Joosen, co-supervisor

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

March 2016

An Application Platform for Multi-purpose Sensor Systems

Klaas THOELEN

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

March 2016

Preface

While this dissertation primarily presents the key results of the research I conducted over the past few years, it also represents the end of an exciting period in my professional life. During this time I was fortunate enough to be assisted and supported by a good number of people.

First, I would like to thank Danny Hughes for being a very approachable supervisor, without whom I might have not even started this PhD. Thanks for the many small and large discussions, and for guiding me throughout the PhD process while also providing freedom to follow a path of personal interest.

My gratitude goes out to Wouter Joosen for initially hiring me as a researcher at DistriNet, and later on giving me the opportunity to capitalize on that experience by pursuing a PhD. Thanks for your insightful feedback and interrupts throughout the years.

I want to thank Sam Michiels and Herman Bruyninckx for being part of my supervisory committee, as well as the additional members of the jury, Christophe Huygens and Steven Latré, for taking an interest in my work and for the interesting discussion during the private defence. Sam, an additional thanks for showing me the ropes of researching throughout my first years at DistriNet. In addition, I would like to thank Joos Vandewalle for chairing the jury.

Thank you to everyone at DistriNet and the department for the enjoyable working atmosphere; in particular all current and former office mates and members of NES. Extra thanks to Nelson and Wouter as pioneers and collaborators on LooCI, and a special mention to Pierre Verbaeten for introducing me to DistriNet. Last but not least, I greatly appreciate all the efforts of the DistriNet project office and secretaries in dealing with the bulk of administration and allowing me to focus on my research.

All of this would not have been possible without the support of friends and family. Thanks for at times wondering what I was doing in Leuven, and at other times taking my mind of it. For different reasons, I want to especially thank ’de Reigers’, tante Lutgarde, and the Espinoza family.

Finally, I am blessed with a great close family around me that is always supportive and around to experience the things that matter; Hans and Ilka, Soetkin and Vinod and Cobi and Mas and Wies, and Kobe. But, most thanks and gratefulness go out to my parents for their unconditional love and endless support!

Klaas Thoelen - March 2016

Abstract

Distributed sensing systems are complex integrations of constrained sensor networks with additional commodity computing platforms. The realisation of such systems requires significant effort in hardware deployment and software development, resulting into a substantial investment to be made. The key complexities that need to be dealt with include a large heterogeneity in hardware and software, and the resource-constraints of wireless sensor networks. To deal with these complexities, early deployments were based on highly optimised and static single-purpose software solutions within the sensor network tier. Research and industry, however, aim to improve the return on investment by shifting towards open sensor networks that provide a reusable infrastructure on which various stakeholders can independently deploy multiple applications, and, where opportune, reuse the functionality provided by others.

The current state-of-the-art however fails to provide a full solution to the open and multi-purpose use of sensor systems. Related work is available on multi-application and -user support, dynamic software configuration, and discovery of application logic; however, none of those technologies provide a well-integrated solution with a high enough level of configurability and openness to result in truly multi-purpose sensor systems. To realise the open use of sensor system infrastructure, additional lightweight middleware solutions are needed that together provide a common substrate across the sensor system tiers, on top of which independently developed application logic can be deployed and reused within various distributed sensing applications. A similar strategy to improve application logic reuse has been put forward by service-orientation; a software engineering paradigm commonly used to build software solutions within corporate back-ends. Its principles and implementations need to be reimagined, however, to be feasible and applicable within the resource constrained environment of sensor networks.

This dissertation presents three contributions towards open and multi-purpose sensor networks. The first contribution facilitates application building across heterogeneous sensor systems; LooCI, or the Loosely-coupled Component Infrastructure, provides run-time configurable abstractions for application modularity and distributed interactions across a range of hardware and software platforms. The other two contributions facilitate the run-time reuse of application logic within open sensor systems. SDlite provides a service discovery solution that takes into account the environmental and operational state of sensor nodes to diversify between the many functionally similar service providers within a sensor network. Lastly, TalkSens provides a framework that facilitates the systematic definition of messages, their coordinated use among multiple parties, and provides lightweight message subtyping support. This improves reuse of third-party application logic and reduces configuration effort. The combination of all contributions results in a highly configurable application platform for open and multi-purpose sensor systems.

Prototype implementations of the presented contributions have been realised on a range of platforms, with a focus on resource-constrained sensor nodes, but including back-end platforms to support multi-tier integration. Evaluation, and the application of these contributions within a smart office deployment, show their feasibility and good performance on constrained sensor nodes. In sum, this greatly advances the paradigmatic shift from static single-purpose sensor systems to more cost-effective dynamic and multi-purpose alternatives.

Beknopte samenvatting

Gedistribueerde sensorsystemen zijn complexe integraties van sensornetwerken met meer traditionele computerplatformen. De realisatie daarvan vereist echter een significante inspanning in hardware installatie en software ontwikkeling, hetgeen resulteert in een substantiële investering van tijd en geld. De belangrijkste complexiteiten waarmee rekening dient gehouden te worden, omvatten de grote heterogeniteit in hardware en software, en de beperkte systeembronnen voorradig in draadloze sensornetwerken. Initiële installaties van sensornetwerken kwamen tegemoet aan deze problemen met behulp van sterk geoptimaliseerde softwareoplossingen binnen het sensornetwerk die één enkele statisch gedefinieerde toepassing ondersteunde. Binnen de onderzoekswereld en in de industrie streeft men er momenteel echter naar om de gemaakte investeringen versneld terug te verdienen door open sensornetwerken te voorzien. Deze doen dienst als herbruikbare infrastructuur en laten verschillende partijen toe om onafhankelijk meerdere toepassingen te installeren, en, indien opportuun, elkaars functionaliteit te hergebruiken.

De huidige stand van zaken binnen het onderzoeksdomein van sensornetwerken biedt echter geen complete oplossing tot dergelijk open gebruik van sensorsystemen. Hoewel gerelateerd werk beschikbaar is ter ondersteuning van meerdere toepassingen en gebruikers, het dynamisch configureren van software, en het zoeken naar herbruikbare applicatielogica, bieden geen van deze systemen een volledig geïntegreerde oplossing met een adequaat niveau van configureerbaarheid en openheid. Om tot werkelijk open sensorsystemen te komen die meerdere doeleinden dienen, zijn er aanvullende lichtgewicht middleware oplossingen nodig die gezamenlijk een gemeenschappelijk substraat aanbieden over de verschillende segmenten van een sensorsysteem. Hierbovenop kan onafhankelijk ontwikkelde applicatielogica geïnstalleerd worden en hergebruikt in verschillende gedistribueerde sensortoepassingen. Een vergelijkbare strategie tot hergebruik van applicatielogica wordt voorgesteld door *service-orientation*; een paradigma binnen de softwareontwikkeling dat veelal toegepast wordt in gedistribueerde systemen voor traditionele serveropstellingen. De voorgestelde principes van service-orientation, en de implementaties daarvan, moeten echter geherinterpreteerd worden om haalbaar en toepasbaar te zijn met de beperkte systeembronnen voorradig in sensornetwerken.

Dit proefschrift stelt drie bijdragen voor die het open gebruik van sensornetwerken voor meerdere doeleinden toelaten. De eerste bijdrage vergemakkelijkt het ontwikkelen van applicaties die gedistribueerd zijn over de verschillende heterogene segmenten van een sensorsysteem. LooCI, oftewel Loosely-coupled Component Infrastructure, biedt dynamisch configureerbare abstracties aan voor modulaire applicaties en gedistribueerde interacties voor verschillende hardware en software platformen. De andere twee bijdragen vergemakkelijken het hergebruik van applicatielogica binnen open sensorsystemen. SDlite laat toe om dynamisch sensorknopen te ontdekken die de gewenste applicatielogica uitvoeren. Hierbij wordt er rekening gehouden met de omgevings- en operationele toestand van de sensorknopen om een onderscheid te kunnen maken tussen de typisch meerdere knopen die dezelfde applicatielogica uitvoeren binnen een sensornetwerk. TalkSens, tot slot, vergemakkelijkt het systematisch definiëren van berichten, het gecoördineerd gebruik ervan tussen verschillende partijen, en voorziet lichtgewicht ondersteuning voor het subtyperen van berichten. Dit laat eenvoudig hergebruik toe van de applicatielogica van een derde partij en verkleint de vereiste configuratieinspanning voor gedistribueerde sensortoepassingen. De combinatie van al deze bijdragen resulteert in een sterk configureerbaar applicatieplatform voor open sensorsystemen die voor meerdere doeleinden kunnen toegepast worden.

Prototypes van de voorgestelde bijdragen werden geïmplementeerd voor verschillende platformen, met een focus op beperkte sensorknopen, maar inclusief krachtigere platformen om integratie te ondersteunen. De evaluatie en toepassing van deze bijdragen in een slimme-kantooromgeving tonen de haalbaarheid aan van de voorgestelde aanpak en de goede performantie op beperkte sensorknopen. Dit levert een significante bijdrage aan de paradigmatische verschuiving van een statisch gebruik van sensornetwerken voor een vooraf gedefinieerd doeleinde naar een meer kosteneffectief dynamisch gebruik voor meerdere doeleinden.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The continuing miniaturisation of computing platforms allows for monitoring of our physical surroundings at an unprecedented scale. Small embedded devices, ranging in size from a few cubic millimeters up to a few cubic centimeters, can now be integrated with the environment and be programmed to monitor a wide range of physical quantities. As monitoring data becomes available in real-time, much more timely reaction becomes possible to events that previously went by unnoticed or were only discovered post-factum.

This (r)evolution gives rise to emerging systems like wireless sensor networks (WSNs), the Internet of Things (IoT) and cyber-physical systems (CPS) that find applications in domains such as supply-chain logistics [13, 45], health-care [4], smart environments [8, 160] and wildlife preservation [85, 112]. According to recent data, the amount of applications and their scale is poised to explode: research firm Gartner estimates that by 2020 the number of small monitoring devices that will be connected to the Internet will increase to 25 billion [55].

This dissertation contributes novel solutions towards the open and multi-purpose application of sensor networks. In this regard, these networks of embedded devices are envisaged as generic computational infrastructure. Such infrastructure hosts various independent applications that can be individually deployed and executed concurrently. This contrasts greatly with the single-purpose and static sensor networks that are typically deployed today.

In this chapter, Section 1.1 provides additional context within which the presented research was performed. Section 1.2 continues with a problem statement, and Section 1.3 introduces this dissertation's key contributions. Finally, Section 1.4 outlines the rest of this dissertation.

1.1 Context

To scope the presented research, the following sections provide background information on wireless sensor networks and their integration within larger distributed computing systems. Section 1.1.3 then continues with a discussion on the transition towards multi-purpose sensor networks and the challenges this introduces. The industrial relevance of the presented research is discussed in Section 1.1.4.

1.1.1 Wireless Sensor Networks

A wireless sensor network (WSN)[1] is a wireless network formed by a number of *sensor nodes*. Sensor nodes are small computational devices that range in size from a few cubic millimeters up to a few cubic centimeters. They are typically equipped with a microcontroller, flash and RAM memory, a radio transceiver and one or more sensors. The latter can be embedded or externally connected and monitor physical quantities like temperature, light, CO_2, sound level, etc. An important feature of most sensor nodes is their limited energy budget. This is typically provided by batteries, although increasingly energy harvesting technologies are also employed. Because of their small form factor and limited energy budget, sensor nodes are considered to be constrained devices with limited computational, communication and memory resources [16]; they typically feature an 8-bit microcontroller, and only a few hundred of kilobytes flash, a few tens of kilobytes of RAM and a networking bandwidth of less than 250 kilobits per second. To prolong their lifetime, efficient energy budgetting is required by minimising computational and communication efforts.

By means of radio connectivity, a set of sensor nodes form a wireless sensor network. The size of these networks can range from only a few up to hundreds, and even thousands, of sensor nodes. They can either be formed in an ad-hoc manner and constituted of sensor nodes only, or include additional infrastructure in the form of a gateway and be connected to a more resource-rich back-end, or even the Internet. Depending on their size, single-hop networks can be formed in which all nodes directly communicate with each other, or multi-hop networks in which intermediate nodes route messages between two interacting nodes that are outside of each other's radio range. Although previously proprietary and special-purpose networking stacks were used, current WSNs typically use traditional networking protocols like IPv6, ICMP and UDP, albeit in combination with dedicated networking protocols for constrained devices like the IPv6 adaptation

[1]Hereafter often referred to as simply *sensor network*

layer 6LoWPAN [123], the RPL [184] routing protocol and the CoAP [156] application protocol.

Wireless sensor networks emerged in their current form in the late 1990's [67, 86, 143]. Since then mature solutions have been developed for most low-level challenges in the operating systems and networking domains. Examples include operating systems like TinyOS [106] and Contiki [38] and the aforementioned dedicated networking protocols. Middleware for wireless sensor networks has been an equally active research domain [61, 126, 176], but has yet to result in widely used platforms and paradigms. Consequentially, application development and management remain difficult and requires specialist skills in software engineering, networking and embedded computing.

While application domains are plentiful, currently the wide-spread adoption of sensor networks is inhibited by the large financial investment they require; sensor network hardware remains expensive and application development and management expertise further increase the necessary costs [118]. Opportunities that can improve the return-on-investment can be found within (i) extended integration of sensor networks with other computational infrastructure, (ii) increased support for application dynamism, and (iii) the multi-purpose use of individual sensor networks. These opportunities are further explored in the following sections.

1.1.2 Integrating sensor networks with back-end systems

As mentioned in the previous section, wireless sensor networks are often not treated as separate islands of networked embedded systems. Instead, their potential is further exploited by integrating them with back-end servers and personal computing devices. In this dissertation, such integrated systems are referred to as *sensor systems*. The larger resources provided by servers enable storage of the gathered environmental data and allow for higher-level information to be derived from that data. Further integration with mobile apps and web-based applications drastically increases the availability of the gathered data to users. A second benefit of such extended integration, is that it also enables the remote control of sensor networks from back-end and mobile devices. This is a primary concern as sensor nodes are often installed in hard-to-reach places; for instance in applications such as river flood detection [70] and volcano monitoring [181].

A characteristic aspect of sensor systems is the large hardware and software *heterogeneity* across the various tiers. An example thereof is depicted in Figure 1.1. Across the sensor, gateway, mobile and back-end tiers, different types of devices are used with very diverse resources in terms of computation, memory,

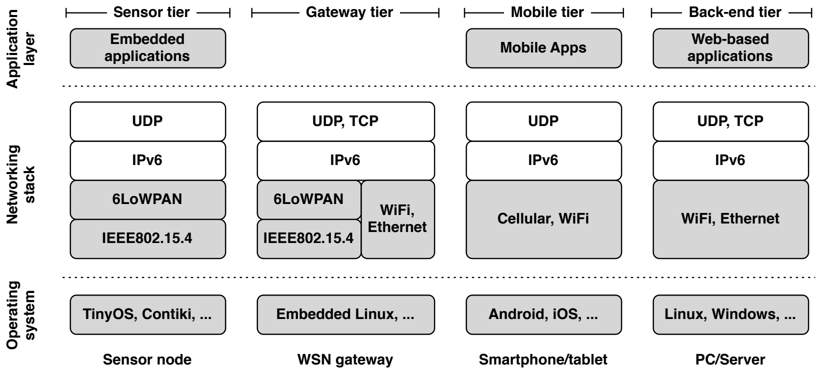


Figure 1.1 – An exemplary overview of the software and networking stacks used in the various sensor system tiers.

communication and energy. While higher layer networking protocols typically shield application developers from the heterogeneity at the lower layers, the diversity in available resources, and the specific nature and behaviour of devices at each tier (e.g. sleeping sensor nodes), result in various specialised software and networking stacks employed across the entire sensor system. Developing distributed applications that span the various tiers therefore requires the expert knowledge of multiple platform specialists.

A number of commercially available solutions support such integration, yet, do so within closed eco-systems only. These systems span all tiers of the sensor system and leave little support for dynamism and flexibility within the sensor network. Instead, sensor nodes simply forward any sensed data to back-end servers, often over a single-hop link within the sensor network. The aim of these systems is to collect huge amounts of data and utilise the powerful computation available at the back-end to make value thereof. Sensolus[2], for instance, employs dedicated devices to track various types of assets (e.g. furniture, cattle, yachts) and seamlessly connects them with cloud-based management support and applications. Systems like IBM Watson IoT[3] and Waylay[4], in turn, provide Platform-as-a-Service (PaaS) solutions to extract higher-level information from huge amounts of heterogeneous data by means of machine learning, and support real-time decision making by feeding the collected data to a rules engine. As discussed in the next section, this dissertation takes a fundamentally different approach to increase the value of sensor networks; i.e. by exploring ways to make

[2]http://www.sensolus.com/
[3]http://www.ibm.com/internet-of-things/watson-iot.html
[4]http://www.waylay.io/

more flexible use of the resource constrained sensor nodes themselves. While on the one hand supporting use-cases that inherently require such complexity within a sensor network (see Section 1.1.4), such support can additionally greatly complement and further increase the commercial value of said available systems.

While integrating sensor networks with back-end and personal computing devices thus greatly improves their usability, such integration is non-trivial as it results in complex distributed systems. Additional development and infrastructural support is required to facilitate and fully leverage this potential within the open use of sensor systems.

1.1.3 Towards open and multi-purpose sensor networks

Early sensor network deployments featured static functionality optimised to perform a single dedicated monitoring task. This fulfilled the initial research purpose of evaluating the feasibility of environmental monitoring with widely distributed and constrained computing devices. Dedicated application logic executed on top of the operating system [125] and extensive care was taken to limit the computation and communication overhead [85, 98, 112, 180, 186]. Software was deployed on nodes prior to their physical installation and remote software deployment was either not possible [85, 180, 186] or only allowed for the installation of complete software images on all nodes in the network [98].

Currently, sensor networks are increasingly envisaged as *multi-purpose* infrastructures [32, 125, 144]. This is expected to improve the return-on-investment as the cost of purchasing and installing hardware can be split over multiple applications and users [102, 163]. Such shared use of sensor infrastructure is however only interesting, and practically feasible, when the various deployed applications can evolve independently. A change to one application may not disturb other applications that are not affected by the change. Therefore, support for *dynamic system configuration* is required, being the ability to extend and modify a system while it is running [95]. Next to adding new functionality, also the available functionality needs to be configurable. This allows for the one-time development of reusable applications that can be adapted to their local context after deployment. A simple example is a temperature sensing application with a configurable sensing frequency to suit multiple use cases. While solutions for dynamic configuration of sensor networks exist, the overview in Chapter 2 shows that these are limited in flexibility.

Additionally, correct management and configuration can only be performed if the initiating stakeholder's view on the system's state is accurate. In a multi-purpose and multi-user environment, such dynamic changes to the system need to be detectable by third-party stakeholders. Support should be provided to *discover* the functionality provided by a sensor node and inspect its configuration settings. This allows for reuse of functionality, if suitable, and prevents duplicate deployment of similar application logic to serve various stakeholders.

In conclusion, great opportunity towards an improved return-on-investment lies in the use of sensor networks as multi-purpose infrastructure. Once sensor nodes are physically installed, this allows for multiple applications to be individually deployed, configured and reused for various purposes. Contemporary application platforms for sensor networks however do not support such extensive flexibility.

1.1.4 Industrial impact

The research presented in this dissertation was driven by industrial needs and evaluated in a number of research projects: MultiTr@ns [78], STADiUM [80], AdMid [75] and COMACOD [76]. All projects featured use cases in the logistics domain. The following section presents the fleet management use-case from the COMACOD project. This is representational for the use-cases in the other projects and clearly shows the need for open and reusable sensor networks as introduced in the previous section.

Fleet management use-case

Today's trucks and trailers are increasingly being equipped with embedded devices to monitor transported goods. For instance, sensor nodes are installed in cooled trailers to monitor the current temperature in the trailer and forward the readings to an on-board unit (OBU) in the truck. An OBU is a small embedded device equipped with an information screen that provides the driver with information regarding its current shipment. It is connected to the transport company's fleet management system and provides the driver with details about transportation tasks, for instance: type of product, pick-up and drop-off locations and monitoring requirements. Additionally, next to forwarding the temperature readings to the transportation company's back-end for further processing and logging purposes, it provides the driver with an overview of the environmental conditions in the trailer.

Integration of the sensor network, OBU and back-end system is however not trivial due to the high level of hardware and software heterogeneity. It is furthermore complicated by application dynamism. After all, across various transportation tasks, trucks connect with various trailers, causing OBUs to interact with different sensor networks over time. Additionally, for each new task, the OBU and sensor nodes need to be (re)configured to precisely monitor the transported goods as required by legislation and customer demands. As such, the sensor network in the trailer functions as a computational infrastructure that is frequently reconfigured to suit the monitoring requirements of each current transportation task.

The dynamism of this use-case clearly shows the need for software evolution in sensor systems as described in the previous section. The full end-to-end monitoring provided by integration of sensor nodes, the OBU and the back-end system, furthermore is a good example of the extreme heterogeneity that needs to be dealt with in a lot of real-world monitoring applications. Specifically to logistics, the systems described in [13, 45] confirm the requirements of the presented use-case.

1.2 Problem statement

As mentioned in Section 1.1.1, the return-of-investment of sensor networks can be increased by their employment as an open and multi-purpose infrastructure that is integrated within a larger distributed environment. The problem that this dissertation tries to solve is how such a multi-purpose sensor network can be provided and made practical use of. This relates to the following set of typical middleware problems projected on the WSN domain;

- *How to deploy new application logic across a sensor system and build distributed applications?*
- *How to discover available application logic in an open sensor network?*
- *How to interact with third-party application logic in an open sensor system?*

Solving these problems is however not trivial due to the specific nature of wireless sensor networks [32, 61, 125]. Features such as restricted resources, increased node mobility, intermittent network connectivity, typical interaction patterns, and limited variation in application functionality render the straight-forward reuse of existing solutions sub-optimal. Most importantly, a balance should be found in the sensor network between added functionality and flexibility on the one hand, and lightweight solutions on the other hand.

In more technical terms, the following key issues need to be solved with the mentioned specific features of sensor networks in mind;

1. Adequate abstractions are needed that provide application-level modularity and support distributed interactions. These need to enable dynamic

composition and configuration of distributed applications across the various sensor system tiers. Hardware and software heterogeneity needs to be abstracted away, as well as low-level system and networking issues.

2. Run-time support is needed to observe the current state of the sensor system and the functionality it provides. This entails mechanisms and abstractions to discover and inspect, possibly third-party deployed, application logic and the context in which it operates.

3. Where application logic of different parties is to interact, additional support is needed to facilitate correct and meaningful communication.

Additionally, the collective solutions to these key issues need to result into an integrated application platform for the open use of multi-purpose sensor systems.

1.3 Contributions

This dissertation presents three novel contributions to the current state-of-the-art in wireless sensor network middleware. Each of them provides a specific solution for one of the issues identified in realising open and multi-purpose sensor systems. These contributions are:

1. Reconfigurable lightweight abstractions for application modularity and distributed interactions provided by LooCI, or the Loosely-coupled Component Infrastructure [71, 72, 73].[5]

2. A status-aware service discovery mechanism provided by SDlite [170, 169] that takes additional non-functional criteria into account.

3. The TalkSens framework, which coordinates the systematic definition of messages, and provides lightweight message subtyping support.

LooCI provides application-level components as a modularisation abstraction and features a distributed event bus that realises loosely-coupled interactions between components. Run-time configuration is supported by means of remote deployment of components, behavioural configuration of component properties, and configuration of local and remote component interactions over

[5]The design and development of LooCI was joint work with two colleagues. Personal contributions focused on (i) LooCI's communication framework, (ii) the LooCI/Sun SPOT implementation, and (iii) unification of configuration APIs in the second release.

the event bus. Additionally, LooCI supports full inspection of components and their interactions, and its various its various implementations enable unified application management across all tiers of a sensor system.

SDLite provides status-aware service discovery that, besides service functionality, also takes the operational and environmental status of service-providing sensor nodes into account. This allows to differentiate between the often multiple sensor nodes providing the same functional service within a sensor network. SDLite features a lightweight mechanism that allows various middleware and system services to share node status information with, amongst others, the service discovery process.

TalkSens is a message definition framework that supports explicit and coordinated definition of messages to facilitate interactions with, possibly run-time discovered, third-party application components. It enables multiple parties to come to a shared agreement of message types and contents, and provides associated development-time and run-time support in the form of generation of message serialisation code, and run-time inspection for message definitions. This facilitates meaningful and error-free interactions between third-party application components. In addition, TalkSens provides message subtyping, which considerably reduces the configuration effort when building distributed applications.

Together, these contributions realise an application platform for open and multi-purpose sensor systems. This integrated solution results in a service-oriented architecture [138] that facilitates development and run-time configuration of multi-tiered sensor applications. The presented contributions are implemented for a variety of hardware and software platforms and are validated through theoretical analyses, quantitative evaluation of their implementations, and practical application in a smart office use-case.

1.4 Overview

A structural overview of this dissertation is presented in Figure 1.2. The chapters in this text are organised as follows:

Chapter 2 provides additional background information about the requirements to the presented research and the selected approach to a solution.

Chapter 3 provides a short high-level overview of an integrated solution provided by the contributions presented in the later chapters.

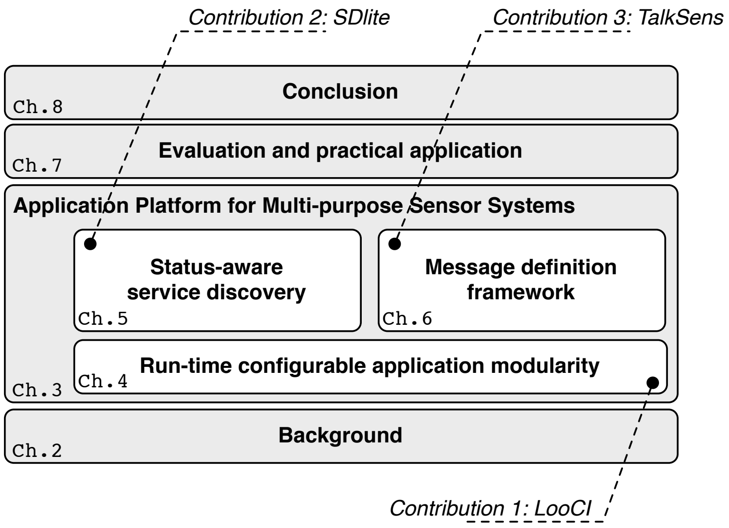

Contribution 2: SDlite
Contribution 3: TalkSens
Conclusion
Ch.8
Evaluation and practical application
Ch.7
Application Platform for Multi-purpose Sensor Systems
Status-aware service discovery
Ch.5
Message definition framework
Ch.6
Run-time configurable application modularity
Ch.4
Ch.3
Background
Ch.2
Contribution 1: LooCI

Figure 1.2 – A structural overview of this dissertation; highlighting the various contributions.

Chapter 4 describes the application and communication abstractions provided by the LooCI component infrastructure.

Chapter 5 presents the status-aware service discovery mechanism provided by SDlite.

Chapter 6 introduces TalkSens, a message definition framework that coordinates the systematic definition of messages, and provides lightweight message subtyping.

Chapter 7 evaluates the design and implementations of the various contributions, and applies them in smart office use-case.

Chapter 8 concludes this dissertation with some notable observations, presents opportunities for future work and provides an outlook on future sensor systems.

Chapter 2

Background

Provisioning support for the open and multi-purpose use of sensor systems requires the integration of a range of software technologies. Features like application modularity, run-time code deployment, remote inspection and configuration, and discovery, all need to be catered for and tailored to the properties of sensor networks. The research presented in this dissertation builds upon the individual solutions to these problems that are presented in the research literature. This chapter presents an overview of the related work in both the WSN and the wider distributed systems domain.

The first part of this chapter surveys WSN-specific technologies that contribute to open and multi-purpose sensor networks. Section 2.1 discusses existing support to serve multiple applications and users, Section 2.2 presents various application configuration approaches, and Section 2.3 overviews discovery solutions. Section 2.4 wraps up and discusses how state-of-the-art application platforms for wireless sensor networks apply these technologies and identifies their shortcomings.

In the second part, Section 2.5 widens the scope and discusses service-orientation. Often successfully applied in more traditional distributed system deployments, it provides inherent support for application dynamism, abstract description of computational logic, and the discovery thereof. Section 2.6 discusses the application of service-orientation within the WSN domain and identifies shortcomings of existing solutions with regard to application dynamism and reuse in multi-purpose sensor networks.

2.1 Enabling multi-purpose sensor networks

While sensor network deployments in the early 2000's featured static and highly optimised functionality [85, 112, 180], the benefits of multi-purpose sensor networks rapidly became understood and explored [188]. This trend has only increased in more recent WSN literature [32, 125] with the emergence of stable solutions for low-level system issues.

This section provides an overview of state-of-the-art solutions that contribute to multi-purpose sensor networks. As shown in Figure 2.1, this can be categorised into multi-application support on the one hand, and multi-user support on the other. The former allows for multiple functional tasks to be concurrently assigned to a sensor network or sensor nodes, while the latter enables various users to concurrently make use of the network. Although both might be supported at the same time, this is not always the case. Section 2.1.1 discusses multi-application support, and Section 2.1.2 presents multi-user solutions.

2.1.1 Multi-application support in sensor networks

Multi-application support in sensor networks can be provided at three levels; the data level, the network level or the sensor node level. Data-level approaches hide away the internals of the sensor network by collecting sensor data at the gateway or back-end and allow multiple applications to make use thereof. At the

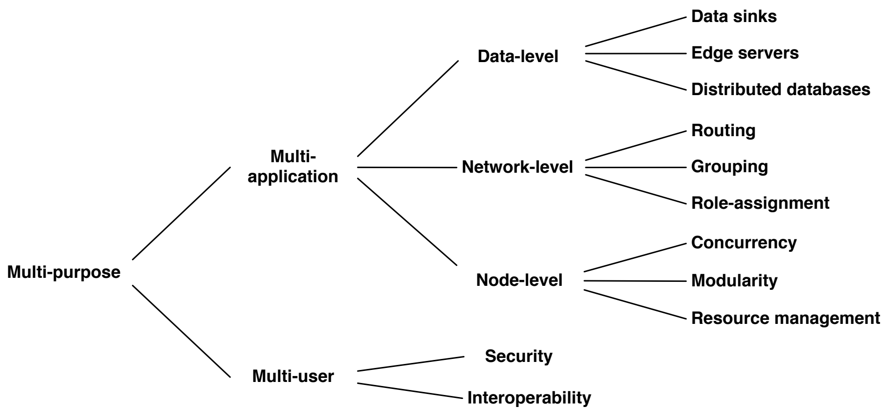


Figure 2.1 – A categorisation of software and networking solutions that contribute to multi-purpose wireless sensor networks.

network level, multi-application support is realised by a differentiation among nodes in the network. Node-level multi-application support, finally, allows a single node to concurrently perform multiple functional tasks. More details, and examples, are discussed in the rest of this section, with a greater focus on the node-level support as it fits more within the scope of this dissertation.

Data-level multi-application support

In this category, the applications reside at the back-end and deal only with the data that is collected at the gateway. In-network details are hidden away by *data sinks*, *edge servers* or *distributed database abstractions*.

In the data sink approach, sensor nodes simply monitor their environment and forward the gathered data to a database at the gateway, which can be queried by multiple back-end applications. The seminal habitat monitoring deployment on Great Duck Island [112] is a prime example of this approach. Edge server approaches apply technologies such as RPC and Web Services to present various back-end applications with a higher-level application interface. In addition to provisioning raw sensor data, additional processing can be performed at the gateway to present higher-level data to client applications. Examples of such systems include Janus [37], Hourglass [158], Global Sensor Networks (GSN) [1] and the Oracle [136] and IBM edge servers [153]. Distributed database abstractions such as TinyDB [110], DSWare [108] and Cougar [185], on the other hand, abstract away the complexities of the sensor network behind a database querying interface. Various back-end applications can query the sensor network via this gateway-based interface. Instead of pro-actively collecting data at the gateway and processing the queries there, the queries are disseminated into the network where the necessary processing and aggregation of data takes place.

While these approaches enable multiple back-end applications to make use of collected sensor data, this is in stark contrast with the limited functionality provided by the sensor network itself; i.e. provide data to the gateway. Individual sensor nodes cannot be directly interacted with and in-network reaction to events is not supported. Furthermore, while back-end applications can be dynamic, the functionality in the sensor network is rather static and leaves little to no room for application specific requirements. When one back-end application requires a change to the logic within the sensor network, this also affects the other applications, with a substantial chance on disturbance. This is not in accordance to the multi-purpose support envisaged in this dissertation.

Network-level multi-application support

Multi-application support at the network level enables various nodes in a sensor network to perform different tasks. An example is provided by a simple wireless sensor and actuator network (WSANs) in which some nodes sense the environment, and upon some detected event trigger other nodes to perform an action. This paragraph discusses *routing*, *grouping*, and *role-assignment* as technologies in support of multi-purpose sensor networks.

In contrast to single-sink routing, unicast and multicast routing protocols allow for communication between two or more arbitrary nodes in the network. This supports more flexible interaction patterns that emerge once different nodes perform different tasks. Although the presented contributions rely upon such routing support to be available, a detailed survey of sensor network routing is outside of the scope of this dissertation. The interested reader is instead referred to the extensive surveys by Akkaya et al. [2], Al-Karaki et al. [3], and more recently, Singh et al. [159] and Watteyne et al. [178].

At higher levels of abstraction, various networking and middleware solutions exist that allow to group nodes together and abstract away the rest of the network to facilitate communication. This allows various groups of nodes to exist in the network, which all serve a different purpose. Overlay networks [58, 102], for example, clearly delineate sets of nodes in a network. Virtual links are established between the respective nodes to isolate network traffic per application. At the middleware layer, neighbourhood programming abstractions aim at simplifying sensor network programming by facilitating data sharing among nodes. In Hood [182] and Abstract Regions [179], physically close-by nodes form a neighbourhood within which data can be easily shared in support of local applications. Logical Neighbourhoods [124] replaces these physical neighbourhoods with a higher-level, application-defined notion of proximity based on logical conditions.

More flexibility in terms of the actual identification of nodes and run-time support therefor is provided by role specification languages and role assignment algorithms [54, 114]. Predicates on node properties are used to describe roles, using a high-level configuration language. Once disseminated into the network, nodes that adhere to such a predicate take on the associated role in support of a distributed application.

In contrast to the gateway systems discussed in the previous section, these approaches bring multi-application support *into* the sensor network. Routing protocols and communication abstractions allow the network infrastructure to be used for multiple purposes. Crucially, for instance, this ensures that nodes serving one application, can be relied upon to route network traffic of other applications. While such features are important, the scope of this dissertation requires additional support for the multi-purpose use of the computational resources on a single node.

Node-level multi-application support

The execution of multiple applications on a single sensor node requires dedicated software for a variety of features; e.g. nodes need to be able to concurrently execute various tasks, the various applications need to be separated from each other and underlying system logic to enable their individual management, and sensor node resources need to be fairly shared among the various applications. The following paragraphs discuss the state-of-the-art in *concurrency models* and *software modularity* within the WSN domain, and briefly touches upon *resource management*.

Concurrency models. The concurrency support in WSN operating systems can be categorised in *event-based*, *thread-based*, and *hybrid* approaches. In the event-based model, programs are implemented as event handlers. These are invoked in response to external or internal events and execute their logic to completion. Event-based concurrency has two main advantages; it maps well with the event-driven nature of environmental monitoring, and has a low memory footprint as they use a single stack that is shared between all concurrent tasks. Therefore, TinyOS [106] and Contiki [38], the most popular operating systems in the WSN domain, in principle apply event-based concurrency. However, while this allows for the easy development of tasks that perform short computations, it is harder to develop long-running computations (e.g. encryption, network packet processing), as those require manual partitioning to avoid missing events [90]. To meet this short-coming, WSN operating systems like MantisOS [12] and RETOS [23] apply a multi-thread-based execution model. However, this tends to increase memory overhead [39, 90] as each thread requires its own, typically over-provisioned, memory stack. Which concurrency model to apply, therefore remains an application specific choice. To accomodate this, Contiki actually uses a hybrid model with an event-driven kernel and an optional multi-threading application library [38], or alternatively, the Protothreads [39, 40] programming abstraction. TinyOS, on the other hand, is purely event-based, yet multiple third-party add-ons have been proposed that provide thread-based support; e.g. TinyThreads [116], TOSThreads [90] and Fibers [179].

Software modularity. The flexibility in application management is greatly improved by the ability to delineate cohesive sets of logic into separately manageable entities, called modules. This allows to manage applications independently of each other and underlying system logic. The following paragraphs discuss the most prevalent software modularity solutions in the WSN domain; *native code modules*, *virtual machines* and *component models*. Their remote reprogramming features are discussed later in Section 2.2.1.

Native code modules provide modularisation that operates directly on top of the operating system without providing any other additional abstractions. Modules are written in native code (i.e. mostly C in practice), and are primarily used to update sensor nodes with additional application logic. Contiki [36], for instance, provides default support for dynamic loading of native code modules. TinyOS, on the other hand, does not; yet, such support is added by FlexCup [113]. A special mention in this regard, is in place for the SOS operating system [62], which uses modularity more extensively than the previous examples. SOS provides a limited kernel, on top of which application and network protocol implementations execute in individually deployable native-code modules.

Virtual machines (VM) fulfil a similar purpose as native code modules but at a higher level of abstraction. Applications are typically written in a higher-level programming language, and leverage abstractions for communication, sensing, data processing, etc. provided by a middleware layer. This allows to express similar logic more concisely than using native code. Additionally, the subsequent translation into byte code contributes to more compact deployable software modules. In this regard, VMs justify the computational overhead caused by the extra indirection during byte code interpretation, by reduced module size and energy cost for their remote deployment. An additional advantage is the increased portability. By abstracting away the underlying platform, the VM presents a standard programming interface across a range of target platforms. Both generic *Java VMs* as well as application-specific *script-based VMs* have been applied in sensor networks.

Sun SPOT [9, 135] and Darjeeling [19] are examples of the first category. Besides the modularisation provided by object-orientation, they allow for applications to be wrapped into loadable modules, respectively called *Suites* and *Infusions*. Sun SPOT and Darjeeling are however targeted at different types of sensor nodes, which influences their design. Sun SPOT runs a Java ME compliant virtual machine directly on the processor of resource rich sensor devices (32-bit 180 MHz ARM processor, 512 KB RAM). Darjeeling, on the other hand, supports a substantial subset of the Java language and is specifically designed for 8- and 16-bit microcontrollers with 2-10 KB of RAM.

Script-based VMs aim to even further reduce code size. Applications are implemented using a scripting language and are to a large extent reduced to a composition of VM-provided services. Although this limits applications to the functionality provided by these services, the latter are efficiently implemented by the VM in native code. Examples of the script-based VM approach are Maté

[104], DAViM [120] and DVM [10]. In contrast to Maté, DVM and DAViM support the concurrent execution of multiple scripts.

Finally, component models provide modularity at a higher-level of abstraction than the previous approaches, and aim to improve software reuse [165]. Software components, often simply called components, are software modules that encapsulate a set of semantically related functions and expose that functionality via explicitly defined interfaces. The latter serve as the component's signature and facilitate (third-party) use of the component while encapsulating (i.e. hiding) implementation details. This is in contrast to the previous approaches, which either do not support interaction between software modules (i.e. script-based VMs and Contiki modules) or only via implicit interfacing, requiring a-priori detailed knowledge of the entry points to those modules (i.e. SOS modules and Java VMs).

Component models specifically aimed at sensor networks include nesC [56], Lorien [145], Remora [166], RemoWare [167], Runes [26, 25], and FiGaRo [127]. NesC and Lorien are used to compose the entire software stack of a sensor node, including both system and application logic. NesC is the component model used to develop TinyOS, while Lorien is a component-based operating environment. All other component models are applied on top of an operating system to facilitate development of middleware and application logic. Some only support explicit modularisation at development-time and compile away component boundaries for the sake of efficiency at run-time. Others retain component boundaries and feature run-time component deployment. Runes, for instance, therefor leverages Java class loading and Contiki's support for native code modules. The latter is also used by RemoWare and FiGaRo, while Lorien provides a proprietary native code module format and loading mechanism.

Resource management. Shared use of a sensor node's resources by multiple applications requires vigilant management of those resources [103]. Important shared resources include memory (RAM and ROM), sensor and actuator hardware and the network. To some extent this is provided by WSN operating systems (e.g. hardware virtualisation and memory allocation techniques) as well as middleware solutions (e.g. resource aware programming models and frameworks). Resource management is however considered outside of the scope of this dissertation. The interested reader is referred to the complementary research by our colleague Javier del Cid [33], which addresses this issue in detail.

In conclusion, concurrency models and modularisation are important enablers for multi-application sensor nodes. Additionally, the explicit interfaces provided by component models facilitate third-party use of software modules. The presented solutions, however, mainly focus on the sensor network tier only and do not provide unified modularity and communication abstractions across the various tiers of sensor systems. Furthermore, additional support is needed that improves third-party use; e.g. run-time discovery of deployed software modules and how to interact with them. The following sections address these issues.

2.1.2 Multi-user support in sensor networks

Wherever multiple users share distributed computational infrastructure, interoperability and security concerns arise. The following sections first discuss application level interoperability within the WSN domain, and then shortly overview security issues before referencing to more specialised literature.

Application-level interoperability

For multiple users to share sensor systems and bilaterally use each others provided functionality, interoperability of application and configuration data is required. This section discusses solutions that aim to (partially) provide such support.

Back-end initiatives like SensorML [17] and the Semantic Sensor Network (SSN) ontology [24] aim to improve the management of sensors and the accompanying volume of generated data. Both provide standard models to describe measurement processes by sensors, encode measurements and derive higher-level information of observations. SensorML is an Open Geospatial Consortium (OGC) initiative towards Sensor Web Enablement (SWE) which facilitates the integration of sensing devices with Web service applications. It defines data encodings that provide syntactic interoperability and Web services that facilitate the storage and retrieval of sensor-related data. The SSN ontology extends this with a semantic compatibility layer, allowing users to operate at abstraction levels above the technical details of format and integration, instead working with higher-level domain concepts and restrictions on quality. Their reliance on XML and Web service technologies prevents to extend their use within the sensor network.

Standardisation efforts have also been made to facilitate interactions between, and with, third-party sensor nodes. ZigBee [191] was an early initiative in this regard. Defined to operate on top of IEEE802.15.4, it proposes a number of application profiles (e.g. home automation and industrial plant monitoring). Each profile defines the devices (e.g. lights, switches, etc.) and functions (e.g. on/off, dimmer, etc.) that are useful for that respective application, as well as I/O data formatting.

To improve integration of constrained devices with back-end IP-based applications, application level protocols such as CoAP [156] and MQTT-SN [162] have been proposed. CoAP, or the Constrained Application Protocol, is specified in IETF RFC 7252. Similar to HTTP, it uses a REST-like approach in which resources, presented using URI's, can be consumed or updated. MQTT-SN, applies a more data-centric communication approach in an effort to overcome the difference in addressing schemes applied between e.g. a ZigBee network and an IP network. MQTT-SN is a variation of the OASIS MQTT (formerly Message Queue Telemetry Transport) specification and defines a light-weight publish-subscribe message protocol for constrained embedded devices.

While CoAP and MQTT-SN define application-level communication mechanisms, they do not specify the contents or data that applications exchange using these protocols. Recent initiatives toward this end include the Open Mobile Application (OMA) Lightweight Machine-to-Machine (LWM2M) specification [134], the Internet Protocol for Smart Objects (IPSO) Alliance SmartObject Guideline [79] and Project Haystack [149]. While at its core a device management protocol, OMA LWM2M defines an extensible object model in which an object (e.g. location object) is a collection of CoAP Resources (e.g. latitude) with a specified identifier, cardinality, type (e.g. decimal), units (e.g. degrees), etc. The IPSO SmartObject Guideline applies the LWM2M object model to describe eighteen 'smart object' types, including a temperature sensor, a light controller, an accelerometer and a presence sensor. Project Haystack, in turn, is an open source initiative to streamline working with data from the Internet of Things (IoT). It proposes the pragmatic use of naming conventions and taxonomies to make sense of IoT data and improve semantic meaning thereof. For a number of applications, including automation, control, energy, HVAC and lighting, it describes a set of IoT devices and the data they provide, including which data values and units of measurement to use. While targeting IoT devices, Project Haystack does not directly aim to be applied on constrained sensor nodes.

Compared to the proprietary application protocols used in early WSN research, these solutions contribute substantially to interoperability in the WSN tier and integration with back-end systems. A number of issues can be identified, however. First, both CoAP and MQTT-SN are to a great extent developed to provide standardised interactions between constrained devices and back-end applications, and less with in-network interactions in mind. For instance, the centralised message broker of MQTT-SN requires that all node-to-node interactions pass by the gateway. Additionally, CoAP typically uses JSON to represent data, which, while feasible, is a rather verbose format for use in a resource-constrained environment. Second, OMA LWM2M, IPSO SmartObject, and Project Haystack aim towards more interoperable data by providing message structures to which applications should adhere. This however provides limited freedom or support to specify application-specific messages. Furthermore, additional support is needed that facilitates compliance to these specifications at development-time. Finally, the presented solutions are mostly recent initiatives and adequate implementations were unavailable for the research presented here. For instance, MQTT-SN and client-side support for CoAP Observe have only been added to Contiki in February 2015.[1] Similarly, the IPSO Smart Objects and LWM2M specifications date from late 2014.

Security support for shared sensor networks

A range of security issues need to be dealt with when supporting open use of shared sensor network infrastructure. While a thorough discussion on security is outside the scope of this dissertation, the following paragraph hints towards existing solutions and refers the interested reader to more specialised literature.

When the sensor network is abstracted away from users and their back-end applications only interact with shared databases or edge servers (see Section 2.1.1), traditional distributed systems security measures can be applied, such as database security [11] and WS-Security [131]. In contrast, where multiple users have direct access to shared sensor network infrastructure, dedicated security mechanisms are needed that are tailored towards the resource constraints and dynamic properties of sensor systems. Security solutions need to be in place that ensure node availability and integrity. Each user must only be allowed to configure and make use of the shared infrastructure up to the level for which it was accredited. Deployment and configuration of functionality must be regulated by security policies that ensure the stability of both the network and individual nodes, in order to prevent disturbing the operation of other users' applications. Therefore node-level access control is required during configuration of the sensor system. Also application data flows need to be secured to ensure integrity of the collected data, prevent malicious actuation, and prohibit eaves-dropping. The research by our colleague Jef Maerien [111] discusses these issues in more depth and contributes to the secure sharing of sensor networks by providing a key-sharing mechanism, secure configuration and application communication, and access control.

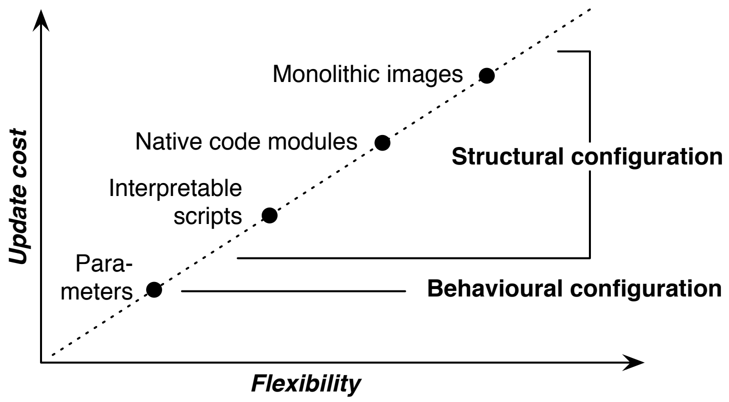


Figure 2.2 – Dynamic software reconfiguration approaches applied within the WSN domain. (Adapted from [10])

2.2 Dynamic software configuration in sensor networks

Using sensor networks as shared and reusable infrastructure requires the individual management of the various applications that execute within the network. This involves deployment of new applications to a sensor node and adjusting their behaviour to the local context. Such changes need to be enacted remotely and at run-time, as they are targeted at pre-deployed sensor nodes.

Support for remote and run-time software changes is provided by *dynamic software configuration*. As shown in Figure 2.2, this can be partitioned in *behavioural configuration* and *structural configuration* [95, 117]. Behavioural configuration changes the behaviour of programs by modifying program variables. It can direct an application to use a different existing strategy, but it cannot add new strategies. Structural configuration, on the other hand, is more course-grained and involves the addition and removal of software functionality. It allows to adapt a system to address concerns that were unforeseen during development.

Similarly, dynamic software configuration in sensor networks ranges from simply adjusting a *parameter* of prior deployed functionality, up to completely reflashing a sensor node's program memory with a new *monolithic software image* [10, 20, 63]. In the following sections, configuration strategies are evaluated in terms of flexibility and update cost. Flexibility in this regard is defined as the level of

[1] `https://github.com/contiki-os/contiki`

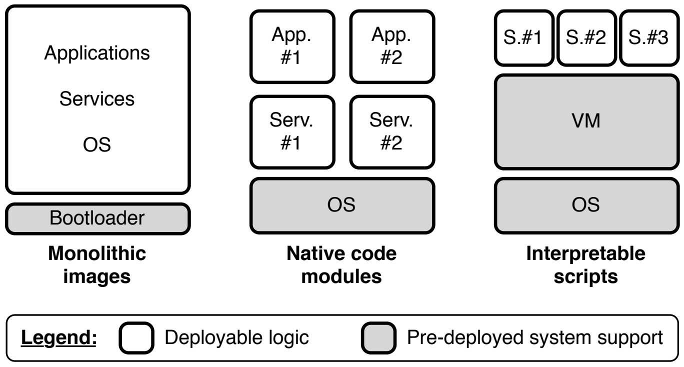


Figure 2.3 – Various structural reconfiguration approaches require varying amounts of run-time system support.

freedom at which software can be reconfigured; e.g. monolithic images allow arbitrary changes to the functionality, while parameters need to be explicitly exposed to be configured. The update cost is defined as the total energy needed to perform a configuration task; this includes both communication and processing of the task.

Section 2.2.1 provides an overview of various structural configuration approaches proposed within the related research literature. Section 2.2.2 does the same for behavioural configuration. Section 2.2.3 discusses the various alternatives in targeting configuration.

2.2.1 Structural reconfiguration

As shown in Figure 2.2, a range of structural reconfiguration approaches are proposed. These provide various degrees of flexibility at various update costs and, additionally, require different levels of run-time system support. An overview of the latter is shown in Figure 2.3 and discussed in the following paragraphs.

Monolithic images. The most flexible form of structural reconfiguration is provided by flashing a sensor node's program memory with a new software image. Software images are monolithic and include both operating system and application code. Consequentially, they are without external software dependencies and thus allow for arbitrary changes. On-node support typically involves algorithms and protocols to recompose the image from individually received fragments and a bootloader to load the new image into program flash.

Image-based software reconfiguration has three main disadvantages. First, they provide an all-or-nothing solution in which a small change or update affects the entire software stack on a node. Second, monolithic images are large in size which incurs a high dissemination cost [150]. And third, data and software state are not persisted across updates as a reboot is typically required. An image update to fix, for instance, a routing bug, therefore not only causes the loss of all routing table entries, but also all other state and application data.

The archetypal example of image-based software configuration in sensor networks is Deluge [74]. While primarily a protocol that reliably disseminates TinyOS images into a sensor network, its implementation enables multiple TinyOS images to be stored on a node and loaded into program flash on demand. To reduce the networking overhead incurred by image-based software configuration, differential update mechanisms have been proposed that only propagate the difference between the current and the new software image [83, 92, 151, 174].

Native code Modules. Module-based software configuration enables modules of native code to be individually loaded on top of a static kernel at run-time. Although some flexibility is lost compared to image-based configuration, the functionality provided by the static kernel is typically limited to basic operating system services like communication, memory management and I/O, leaving great freedom for application functionality. Individual modules represent cleanly separated coarse-grained units of functionality. They are deployed independently and do not disrupt each others execution, nor require a reboot. Usually, mechanisms like dynamic linking and code relocation are required to correctly load a module into the local address space and enable interaction between modules and the kernel. While this increases the complexity of loading a new module, the overall update cost is small since less code needs to be transfered over the network.

Module-based software configuration is by default supported by the SOS [62] and Contiki [38] operating systems. Both use a proprietary compacted Executable and Linkable Format (ELF) to transfer modules. A variant of module-based software configuration is provided by Java virtual machines like Sun SPOT [9, 135] and Darjeeling [19]. Here, optimised and byte-code-verified binary bundles are loaded into a Java virtual machine instead of on top of an operating system kernel.

Interpretable scripts. Interpretable scripts present the most cost-effective, but least flexible type of structural reconfiguration. As discussed in Section 2.1.1, such scripts merely invoke natively implemented services that are provided by a virtual machine through a high-level API. While restricting application flexibility to the functionality of the provided services, this is considered sufficient as the application purpose of sensor networks in general is also limited in range.

The resulting byte-code scripts can be very concise in contrast to native code modules or images. While this decreases the communication overhead during configuration, the added indirection during run-time interpretation requires greater computational effort. Consequentially, this technique is primarily suited when frequent reconfiguration is needed as in those cases the reduced communication overhead is greater than the additional computation overhead.

Maté [104] first explored the use of virtual machines in sensor networks. By providing a high-level interface, it simplifies programming and allows complex programs to be very short (under 100 bytes). As programming against a fixed high-level interface can be found restrictive, more flexibility is provided by ASVM [105], DVM [10] and DAViM [120]. ASVM extends Maté with compile-time customisation of the byte code instruction set, while DVM and DAViM support run-time updates to the instruction set of their respective VMs.

Additional flexibility can be achieved through the combined application of the above approaches. As mentioned, DVM and DAViM achieve this by combining the modular and scripted approach. Similarly, Deluge can be applied to dynamically alter the underlying kernel of modular systems.

2.2.2 Behavioural reconfiguration

The behaviour of functionality on a sensor node can be fine-tuned through the reconfiguration of parameters. The level of flexibility is low and determined by the parameters exposed by applications or system services. On the other hand, such reconfiguration can be performed at a low cost, since it only requires the dissemination of a new parameter value. Run-time parameter configuration requires support thereto from supporting middleware or the applications themselves (see Section 2.4.3). A few approaches however have tried to provide parameter reconfiguration in a more generic manner.

The Sensor Network Management System (SNMS) [173] is a light-weight application-agnostic management system that allows programmers to easily expose attributes of TinyOS components over a multihop network. A dissemination protocol allows to query and configure all exposed parameters, while a collection protocol supports run-time state monitoring.

Marionette [183] provides the ability to call functions and read or write variables of pre-compiled embedded programs at run-time. Based on annotated code, the Marionette compiler generates both on-node code and a back-end Python client tool. The client tool allows to peek and poke variables and invoke function calls via an RPC interface. While primarily aimed at interactive development and debugging, Marionette can be used to remotely configure the entire software stack.

In Chi [49], system configuration is separated from system logic by extracting configuration parameters of different system services into a central registry, called a blackboard. Configuration policies operate on the configuration data on the blackboard to optimise system performance. While this allows for great configuration flexibility across the software stack, Chi does not support remote interaction with the blackboard.

2.2.3 Reconfiguration targeting

An orthogonal concern to the functional granularity of reconfiguration tasks, is the dissemination thereof. Distinctive factors in this regard are *target-selection*, *reliability* and *efficiency*. Within the application level scope of this dissertation, primarily target-selection is of importance.

Target-selection defines which nodes are to be reconfigured. Existing reconfiguration approaches either target (i) the entire network, (ii) a group of nodes, or (iii) an individual sensor node. The functional variation within a network often determines which targeting is the most suitable. Network-wide reconfiguration, for instance, is more efficient in a functional homogeneous network than sequential individual configuration of all nodes. Additionally, a combination of targeting might be beneficial for different configuration tasks; e.g. network wide code deployment and individual parametrisation. The multi-purpose sensor networks envisaged in this dissertation require the possibility to target individual sensor nodes for configuration. While WSN research often favours network-wide reconfiguration [164, 177], also more finer-grained targeting is supported.

Deluge [74], for instance, is used to reliably disseminate TinyOS images across a sensor network. Other monolithic approaches such as XNP [84] and MNP [96] support updating a single node or a group of nodes, respectively across a single hop and multiple hops. Modular updates, on the other hand, are mostly performed on more finer-grained targets. The native code modules of Contiki and SOS, as well as the Java applications of Sun SPOT and Darjeeling are for instance deployed in a unicast manner. FiGaRo [127], on the other hand, builds upon a neighbourhood abstraction to disseminate Contiki modules to a group of sensor nodes. Most VM approaches like Maté [104], DVM [10] and DAViM [120] use efficient code propagation mechanisms, like Trickle [107], to distribute small-size scripts to all nodes in a network. SensorWare [18], on the other hand, supports the deployment of scripts to individual nodes, or a group of nodes that comply to an attribute-based description. Finally, Marionette [183] and SNMS

[173] provide parameter configuration of all nodes in a sensor network, with the former also supporting configuration of individual nodes.

2.2.4 Conclusion

Dynamic software configuration received substantial attention in the WSN community. Various approaches thereto primarily aim at facilitating debugging, adaptation to a local context, and repurposing of sensor nodes. Their application in support of open and reusable sensor networks is less frequent, although, the proposed solutions do provide contributions thereto. From the various structural configuration methods discussed in this section, particularly native code modules suit the requirements of multi-purpose sensor networks. They provide greater flexibility than interpretable scripts, while having a lower update cost than monolithic images [150]. Multiple modules can share underlying system services and operating system functionality, yet these provisions do not impose large restrictions on their own functionality and, for instance, allow low-level interactions with sensor hardware. As discussed in Section 2.1.1, they can be used to encapsulate components, which by means of exposed component properties provide additional support for behavioural configuration. In short, further integration of various approaches holds great promise of achieving the flexible application configurability envisaged in this dissertation.

2.3 Discovery of application logic

Third-party use of application logic in a multi-purpose sensor network, requires that logic to be discoverable at run-time. More precisely, the presence of that logic needs to be determinable as well as the interface via which it can be invoked; i.e. the provided functions or messages handled. Such application discovery comprises two complementary features; (i) an abstract *description* that provides details on both the semantics and interface of an application, and (ii) a discovery *mechanism* that allows to query for such functional descriptions. This section surveys functional representation and run-time discovery approaches proposed in the relevant literature. How discovery requests are efficiently disseminated into a sensor network and replied to, is considered outside the scope of this dissertation. The interested reader is referred to the survey by Anwar et al. [5].

Static single-purpose deployments often lack the need for application discovery. Additionally, low-level embedded programming and constrained resources render the run-time availability of explicit functional descriptions to be less important. Therefore, the only representations that are mostly available in WSN programming solely serve development time purposes. They are provided by language-constructs, such as function definitions in Contiki and nesC component interfaces in TinyOS. This prevents their run-time discovery and by consequence third-party reuse of the represented logic.

Improved support thereto is provided by run-time configurable component models. Examples include RemoWare [167], FiGaRo [127] and Lorien [145]. Components specify their provided interfaces and dependencies, which allows the supporting run-time system to perform dependency resolution upon component deployment. While in RemoWare and FiGaRo this is done merely based on interface naming, Lorien provides more type-safety via hash-codes that are based on the entire interface type including parameter types and return type of all function prototypes. In effect, only semantic representations are thus available at run-time and no complete interface descriptions.

Reflective component models like OpenCOM [27] and Runes [26] provide more details. Full interface representations, including function name and parameter and return types, are inspectable at runtime to allow dynamic invocations of previously unknown, interfaces [27]. Reflection is however mostly applied to provide self-adaptive systems that autonomously reconfigure themselves to adapt to detected context-changes; for instance, switch to a more reliable and performant network configuration upon disaster detection [70]. Such discovery and representation of functionality are constrained to the local run-time and not available to instructed reconfiguration by an external third party. As a result, while more generic interface descriptions, for example using CORBA IDL [133] in OpenCOM, might be available at development time, run-time available interface representations are often much tied to the low-level details of the implementation platform (represented as Java classes, C-structs, etc.) While this allows the run-time system to reason over them, these interface descriptions are (i) platform-specific and (ii) not externally available.

A more open, platform-independent description of functionality is provided by the Constrained RESTful Environments (CoRE) IETF group in the context of CoAP. The proposed Link Format [154] supports resource discovery of CoAP resources. The main function of such a discovery mechanism is to provide Universal Resource Identifiers (URIs, called links) for the resources hosted by a sensor node, or *server* in their terminology. A well-known relative URI `/.well-known/core` is defined as a default entry point for requesting the list of links about resources hosted by a server (i.e. a CoAP enabled sensor node). The link format provides a number of attributes that can semantically type a resource and provide an interface description.

In addition to on-line discovery, CoRE also proposes off-line discovery in the form of a Resource Directory [157]. Such a directory is a back-end service that hosts descriptions of CoAP resources held on sensor nodes. This supports situations where direct discovery of resources is not practical due to sleeping nodes or disperse networks. Sensor nodes are assumed to proactively register and maintain resource directory entries on the Resource Directory, which are soft state and need to be periodically refreshed.

In conclusion, some support for interface descriptions and their run-time discovery has been presented within the context of wireless sensor networks. These approaches however mainly support local adaptation and do not allow for the remote discovery by a third party. A recent exception is provided by CoAP, which however is only an application protocol that requires further integration within an (existing) application framework. Discovery of application logic within a service-orientated context is further discussed in Section 2.5.

2.4 State-of-the-art open and multi-purpose support

The previous sections reviewed a range of operating system and middleware solutions that can contribute to the concurrent execution, run-time configuration and open use of applications in sensor networks. This section revisits the requirements posed in Section 1.2 and critically evaluates how state-of-the-art sensor network research provides support for their fulfilment. To provide a clear overview, the discussion is limited to solutions that support multiple concurrent applications and their run-time deployment. Besides a selection of the previously discussed solutions, a number of representative application frameworks are reviewed that integrate those solutions.

2.4.1 Evaluation criteria

Table 2.1 evaluates the selected approaches along a number of criteria that reflect the requirements to open and multi-purpose sensor systems. These are categorised as follows.

Run-time software configuration. Software configuration at the structural and behavioural level is required at run-time. Structural configuration is categorised in the provided modular abstraction and the type of interactions between the latter. Modularity is either provided by native code *modules (m)* or interpretable *scripts (s)*. Interactions between modules (i.e. either native code modules or interpretable scripts) have a *local* scope *(l)* when allowed only between modules on the same sensor node or a *distributed* scope *(d)* when allowed between modules on different nodes. Run-time configuration of these interactions is of importance for flexible reuse and is either (i) *not supported (o)*, allowing only static interactions at run-time, (ii) *internally* supported *(i)*, which allows only the underlying run-time environment to reconfigure interactions in an automated manner, or (iii) *externally* supported *(e)*, allowing an external third party to reconfigure interactions. Behavioural configuration allows for fine-tuning the deployed software modules to their run-time and environmental context by means of changing fine-grained property values. This is either *not supported (o)*, allowed only *internally (i)* by the supporting run-time environment, or *externally (e)* by a third party. The last criteria in terms of configuration is targeting. This reflects whether configuration actions can be performed on a *single* node *(nd)*, a *group* of nodes *(gr)* or the entire *network (nw)*. Depending on the criteria that determine group membership, the amount of nodes in a group can range from none up to all nodes in the network.

Run-time discovery. The run-time discovery of the deployed functionality and its operational context is required to enable third-party reuse. This entails both a representation thereof, as well as a mechanism that allows to retrieve said representation. The functional representation might describe only the *semantics* of the provided functionality or can provide more *structural* details with regards to its interfaces and how to invoke them. These various representations are either (i) *not provided (o)*, (ii) provided in a *simple* or naive manner *(s)*, or (iii) provided in a *coordinated* manner *(c)* that facilitates third-party reuse. Simple semantic descriptions are given by naive and ad-hoc naming, while a coordinated approach provides a more reusable model-based solution. Simple structural descriptions are low-level and platform-dependent, while coordinated structural descriptions are independent of the implementation platform. Besides such functional information, also operational and environmental information can be included during discovery. Such context information is either *not included (o)*, or *included (x)*. The final discovery criteria deals with the scope within which functionality can be discovered. There is either (i) *no support (o)* for discovery, (ii) only *internal* support *(i)* for local discovery within the underlying run-time environment, or (iii) *external* support *(e)* that allows third party users to discover the functionality provided by a sensor node.

Operating system heterogeneity. Sensor applications that span the sensor network, mobile, and back-end tiers require an application platform that abstracts away the different underlying operating systems. Such heterogeneity is either (i) *not provided (o)*, (ii) only provided within a *single tier (x)*, or (iii) provided across *various tiers (+)*.

	Run-time software configuration					Run-time discovery				OS heterogeneity
	Structural configuration			Behavioural configuration	Targeting	Representation			Discovery scope	
	Modularity	Interactions				Semantic	Structure	Context		
		Scope	Configurable							
	o/m/s	o/l/d	o/i/e	o/i/e	nd/gr/nw	o/s/c	o/s/c	o/x	o/i/e	o/x/+
Requirements	**m**	**d**	**e**	**e**	**nd**	**c**	**c**	**x**	**e**	**+**
Operating systems										
Contiki [38]	**m**	o	o	o	**nd**	o	o	o	o	o
SOS [62]	**m**	l	o	o	nw	o	o	o	o	o
Virtual machines										
Sun SPOT [9]	**m**	l	o	o	**nd**	s	o	**x**	**e**	o
DVM [10]	**s/m**	l	o	i	nw	o	o	o	o	o
DAViM [120]	**s/m**	l	o	i	nw	o	o	o	o	o
Component models										
Runes [26, 25]	**m**	l	i	i	**nd**	s	s	o	i	**+**
FiGaRo [127]	**m**	l	i	o	gr	s	o	**x**	i	o
Lorien [145]	**m**	l	i	o	**nd**	s	s	o	i	o
Remora/RemoWare [166, 167]	**m**	l	i	i	**nd,nw**	s	o	o	i	o
Integrated application frameworks										
Melete [189]	s	o	o	o	gr	o	o	o	o	o
Agilla [50]	s	**l,d**	o	o	gr	o	o	**x**	o	o
SensorWare [18]	s	**l,d**	o	x	**nd,gr**	s	o	**x**	o	x
TinyCubus [114]	**m**	l	i	i	gr	?	?	**x**	i	o
SenShare [102]	**m**	**d**	i	o	gr	s	o	**x**	o	o

o/m/s: o = none, m = module, s = script - o/l/d: o = none, l = local, d = distributed - o/i/e: o = none, i = internal, e = external - nd/gr/nw: nd = node, gr = group, nw = network - o/s/c: o = none, s = simple, c = coordinated - o/x: o = no, x = yes - o/x/+: o = none, x = within a single tier, + = across tiers

Table 2.1 – Overview of the support provided by state-of-the-art sensor network solutions towards the realisation of open and reusable sensor systems. (Criteria evaluations that fulfil the proposed requirements are in bold.)

2.4.2 Operating systems, virtual machines and component models

The state-of-the-art overview commences with an evaluation of a number of WSN operating systems, virtual machines and component models.

Contiki [38] and SOS [62] are the most prominent WSN operating systems that provide inherent support for run-time configurable concurrent applications. Both allow for run-time deployment of binary modules but provide no inherent support for additional configuration, nor discovery. While Contiki modules can only directly interact with the kernel, SOS modules can invoke functions of each other or exchange events. As such, only local and static interactions are supported. Being dedicated WSN operating systems, they do not support heterogeneity across the multiple tiers of a sensor system.

The Sun SPOT [9] platform provides an operating environment by means of a Java ME virtual machine that runs directly on the micro-controller of a dedicated, resource rich sensor node. Multiple Java applications can be individually deployed. They execute in isolation and interact locally via a RPC-like mechanism. Local discovery of interfaces is supported based on interface names, yet such discovery needs to be hardcoded in client applications. Additionally, a back-end API allows the remote discovery of applications as well as a set of system state variables. No interface representations are provided during discovery, which limits third party reuse of deployed applications.

DVM [10] and DAViM [120] are two SOS-based virtual machines. Both allow multiple concurrent applications in the form of run-time deployable scripts, as well as dynamic updates of the instruction set of the VM by means of binary modules. Although scripts cannot directly interact, local interactions between scripts and modules are supported. Amongst others, this allows for run-time behavioural reconfiguration of deployed modules to better suit the application context. Both scripts and modules can only be disseminated over the entire network, which limits functional diversity across sensor nodes. No discovery support is provided.

Representative run-time reconfigurable component models for sensor networks include Runes [26, 25], FiGaRo [127], Lorien [145] and RemoWare [167]. All four use dynamically loadable binary modules to implement components. Their explicit component interface definitions allow for configuration of component interactions. Only local interactions are supported, however, although remote interactions can be implemented in a dedicated component. Runes provides the most configuration flexibility via dedicated connectors that allow to explicitly bind components together. These configurations can however only be actuated from within other components. The same limitation holds for behavioural configuration. Component interactions in FiGaRo, Lorien and RemoWare, on the other hand, can only be configured based on the dependencies they describe and which are resolved upon component deployment. RemoWare applies the same approach for behavioural configuration; component properties can be configured only at deployment time, for instance to retain state during the update of a component. While all four component models apply arbitrary naming as a semantic representation of components or their interfaces, Runes and Lorien additionally provide structural interface descriptions. In Runes this is provided by language constructs (i.e. Java interfaces or C structures that contain function pointers), while Lorien provides a hash value based on the entire interface type, including parameter types and return type. RemoWare, in turn, does provide a standardised XML-based SCA interface description, however this is only a development time attribute. Discovery of functionality is only supported locally, often in support of dependency resolution. A description of the state and resources of a node is only provided by FiGaRo, where it is used to determine whether a node belongs to the group of nodes to which a component is being deployed. Component deployment to an individual node and the entire network are also supported by the other component model approaches. Finally, with implementations for Contiki, Java and C/Unix, only Runes provides implementations that span across the various tiers of a complex sensor system.

2.4.3 Integrated application frameworks

This section continues the state-of-the-art overview with an evaluation of a number of integrated application frameworks. As they are first mentioned here, these frameworks are described in more detail.

Melete [189] realises multi-application sensor networks by means of group-based deployment of interpretable scripts. Each node provides an adapted Maté virtual machine that supports the concurrent execution of multiple scripts. An application corresponds with a group of sensor nodes that each individual node can dynamically join and leave based on its contemporary local status (i.e. sensed data, node properties). Scripts can not directly interact with each other and only basic networking functionality is provided by the VM. They can also not be individually configured and do not provide an externally available representation.

Agilla [50] explores the benefits of using mobile agents and tuple spaces as a foundation for multi-application sensor networks. Sensor networks are initially deployed without an application installed, after which users reprogram the network by injecting (diverse) agents into the network. Agents implement application behaviour in Maté-alike interpretable scripts. They interact locally in a loosely-coupled manner via a tuple space. Additional support for distributed querying of remote tuple spaces is also provided. These interactions are not configurable however, nor is there support for the discovery of agents. The context of each node is important and is available via the tuple space to guide migration of agents across the network. Only TinyOS-based nodes are targeted by the Agilla implementation.

SensorWare [18] aims at improving sensor network programming through the automated dissemination of scripts into a network that together perform distributed algorithms. Each node can run multiple scripts that make use of sensing, communication and computation services provided by the middleware. Additional supporting services can be deployed to a node at runtime in the form of control scripts. Scripts are identified using simple naming and identifiers. They can be parameterised to be tailored to the local context or to retain their state when migrating between nodes. Interactions between scripts occur both locally and remotely via a mailbox abstraction. Node-level configuration targeting is supported, but configuration is to a large extent autonomous in order to realise the distributed algorithm. SensorWare provides no support for discovery of provided functionality, yet other node's context can be retrieved via mailbox interactions in support of script migration. Finally, while its Tcl and Linux-based implementation theoretically allows it to be deployed on a range of devices, only a PDA deployment is described. This is however not suited for constrained sensor nodes.

TinyCubus [114] provides an adaptable software architecture in support of heterogeneous applications and their dynamic requirements in TinyOS-based sensor networks. It provides several implementations of system services and based on provided meta-data selects the most appropriate one to serve the current application(s). System services and applications can be deployed at run-time using the FlexCup [113] mechanism. Local interactions take place via a state repository to facilitate cross-layered sharing of data and via function invocations. To facilitate the latter, local interface dependency resolution is executed when new functionality is deployed. While the state repository allows for flexible local parametrisation of components, such functionality is not offered to external entities. Multi-application support at the network level is provided by classifying nodes into different roles based on their state or context. These roles also determine the targeting of component deployment. While it is unclear how component and interface semantics and structure are represented, they are required in some form for run-time dependency resolution.

SenShare [102] addresses the challenges of protecting and isolating multiple applications both inside a node and a network. A hardware abstraction layer enables multiple TinyOS images to operate on a single node, thus providing a modular reprogramming solution. Deployment targeting is performed via SQL-like commands that identify the targets of a modified Deluge dissemination action. SenShare applies overlay networks to isolate communication per application. While this enables to dynamically determine the nodes that together implement an application, it does not enable remote configuration of those interactions. Numerical application identifiers determine overlay network membership and a limited range of context variables (location, sensing modalities and available resources) are taken into account during deployment target selection. Dynamic discovery of applications is however not provided. SenShare targets rather high-end sensor nodes that are able to concurrently execute multiple TinyOS images. It is only implemented on top of an embedded Linux version for the resource-rich iMote2.

2.4.4 Conclusion

The presented overview shows that many state-of-the-art WSN research efforts partially share the requirements posed in this dissertation. The focus is however often on a different need [122]; e.g. self-organisation (Melete, Agilla, SensorWare), quality-of-service (TinyCubus, SenShare), context-awareness (Melete, Agilla, TinyCubus), and programmability (Melete, Agilla, SensorWare). Consequentially, support for configurability is only provided within closed sensor systems.

Support for modularity seems to be well established, yet, the interactions between deployed software modules are still often only locally defined, and at times static. Additionally, the mostly single-platform implementations complicate the integration of sensor networks within wider distributed sensing applications. Furthermore, while behavioural configuration is provided by a considerable number of platforms, this is mostly in support of automated local adaptation. While certainly beneficial in the envisaged large-scale use of sensor networks, such configuration support should also be provided in an external manner so that third-party clients or software can aid in these configurations. The same conclusion can be made with regards to the discovery of deployed functionality. While a number of solutions provide simple functional representations, and even contextual information, these are not detailed enough and restricted to internal use in support of automated behaviour and configuration. This limits the need for platform-independent representations based on a commonly agreed upon model. Yet, such support is of great importance for open sensor systems. While considerable key features of open sensor networks have thus been realised, additional effort is needed to fine-tune those features and improve their integration.

The following section discusses service-orientation; a software engineering paradigm that inherently supports loose-coupling of software modules and discovery in a platform-neutral manner. This holds great promises to further enable more openly usable and configurable sensor systems.

2.5 The Service-oriented approach

This section widens the scope to beyond dedicated WSN research, and investigates the features of *service-orientation* [138]; a software engineering paradigm that provides solutions that map well to the challenges defined in this dissertation (see Section 1.2). The following sections shortly introduce the main principles of service-orientation, and describe how these are influenced by the specific context of sensor systems.

2.5.1 Service-orientation: Architecture and principles

Service-orientation emerged as a reaction to the shortcomings of the earlier silo-based approach of (distributed) application building [42]. In the latter, all functional requirements are straight-forwardly realised in solution logic. While this can result in efficient solutions, across multiple applications it leads to redundancy at both development-time and run-time; considerable amounts of supporting functionality needs to be repeatedly implemented, and executed, across those applications. Furthermore, as each application is build in isolation, future integration is complicated and requires considerable development effort.

To alleviate these shortcomings, service-orientation builds upon the principles of *separation of concerns* and *encapsulation*. As such, well-defined blocks of functionality are implemented as *services*, which form the fundamental unit of service-oriented logic. While each service deals with the requirements of a specific piece of the application, they are intended to be implemented in a generic and reusable manner. This allows to use them repeatedly across various applications which compose multiple services together. To foster this reuse, service descriptions provide an abstract description of the technical interface of a service. Such a description provides details on how to invoke the service's functionality. As their interface descriptions are programmatically interpretable, services can be discovered at run-time.

Service provisioning is organised via a *service-oriented architecture* (SOA) [138], shown in Figure 2.4. *Service providers* publish their services to the *service registry* by providing the service description and a link to a respective service

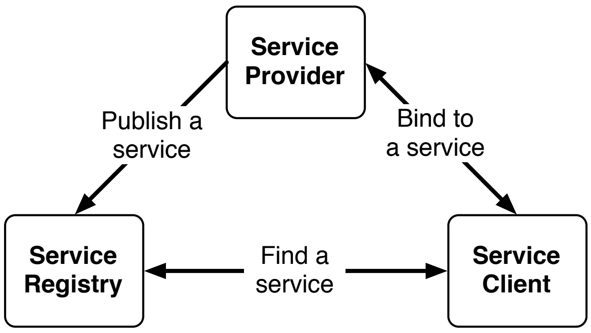


Figure 2.4 – The basic service-oriented architecture [138].

instance. This allows a third party, the *service client*, to query the service registry for a certain service and be informed on where it is available and how to invoke it. As a result, instead of implementing applications entirely from scratch, prior developed services are composed with application-specific ones, to jointly provide the required functionality. Such late-binding of services provides loose-coupling, which, in theory, allows independent adaptation of services without disturbance of the others. Finally, service descriptions should provide information on the functionality of a service, but not on its implementation and underlying technology. As such, they are platform-independent and abstract away underlying platform heterogeneity.

The sole application of a SOA, however, does not necessarily result in the increased reuse of services. It only provides a setting in which such reuse can be achieved. At design time, services should be envisaged as participating in multiple run-time compositions, even when no immediate composition requirements exist. This, however, requires fine balancing and one should prevent to design overly generic and hard-to-use services [42]. While beneficial in the long term, this tends to increase complexity, cost, effort and time to build software at a short term.

Beyond the basics: Extended SOA

Additional concerns, not dealt with by the original basic SOA, discussed in the previous section, have been dealt with in what is commonly referred to as the *extended SOA* [139, 140]. This provides extensions that improve support for composition and coordination of services, and conformance and quality-of-service requirements. On the other hand, concerns such as efficiency, responsiveness and reliability are dealt with to furnish business processes and enable cross-enterprise service compositions. While of great importance to the successful realisation of service-orientation to automate business processes, this dissertation is mainly concerned with the properties of the basic SOA. A number of publications from our research group [33, 68] have however explored the opportunities of these extensions in a WSN context.

2.5.2 A case study: Web services

Service-orientation is a technology-neutral paradigm and has been implemented on top of several software technologies like Jini [7], CORBA [94], Apache Thrift [6]. It is however often erroneously identified with its most common implementation; i.e. Web services. A Web service is a network available service that implements solution logic, is identified by a Uniform Resource Identifier (URI) and accesible using Web technologies. The World Wide Web Consortium (W3C) identifies two major classes of Web services [15]: Arbitrary Web services and REST-compliant Web services.

Arbitrary Web services expose an arbitrary set of operations and build upon standard Internet protocols. The operations supported by such a Web service are described using the Web Services Description Language (WSDL). Service interaction typically occurs using SOAP (originally Simple Object Access Protocol) messages over HTTP (HyperText Transfer Protocol). These technologies rely upon the eXtended Markup Language (XML) to standardise message contents. Further composition and management support is provided by WS-* extensions for security (WS-Security), reliability (WS-ReliableMessaging), trust (WS-Trust), policies (WS-Policy), etc. The WS-* software stack is to a large extent standardised and provides good support for complex operations. This, however, comes with a high-learning curve to developers and heavyweight implementations. Its frequent application of XML additionally causes communication and parsing overhead due to verbose content formatting in exchanged messages. Although some implementations exist for sensor networks (see Section 2.6.4), Arbitrary Web services are generally considered to be too heavy-weight for constrained sensor networks [97, 101].

Recently, more lightweight REST-compliant Web services have emerged. Unlike their arbitrary counterparts, these do not build on top of a large set of standards to provide uniformity, yet instead apply the REST architectural style [48]. Here, a web services consist out of a collection of pieces of information, called resources, that each are identified by a unique global identifier (i.e. URI). These resources can be interacted with in a RESTful manner using a limited set of stateless operations (i.e. GET, POST, PUT, DELETE). Message exchange occurs with the use of XML or, alternatively, with the less verbose JSON (JavaScript Object

Notation) format over HTTP. In general, REST-compliant Web services are better suited for simple and lightweight applications. They do however require additional effort to support more complex interactions, due to the absence of supporting standards. RESTful communication has received considerable attention in sensor networks through CoAP, as discussed earlier in Section 2.1.2, and in Section 2.6.3.

2.5.3 Conclusion

Early static sensor network applications exhibit great similarities with the silo-based applications that service-orientation tries to move away from. Consequentially, service-orientation provides solutions that fit well with the challenges identified in support of open multi-purpose sensor systems, discussed in Section 1.2. These solutions include abstract service descriptions, service discovery and loose-coupling. On the other hand, sensor systems exhibit a number of characteristic features that diversify them from more traditional distributed computing environments. Consequentially, the application of service-orientation in sensor systems requires a re-evaluation of its concepts and principles. The following section, revisits the fundamentals of service-orientation in the light of sensor systems, and explores its application in WSN research.

2.6 Service-orientation and sensor networks

The application of service-orientation within a WSN context enables sensing and data gathering functionality to be abstracted as reusable and platform-independent services. Together with abstract service descriptions and service discovery, this can greatly facilitate building distributed sensing applications. The WSN research literature reports on a range of specific reasons for the application of service-orientation; these include interoperability [152, 101, 47], reuse [152], heterogeneity [51, 101, 89], adaptation [52, 101, 46, 47], ease of development [46, 47, 97, 22], scalability [122] and automation [119].

However, while the concepts of service-orientation remain valid, their implementation needs to be reimagined to suit the characteristics of sensor networks. Section 2.6.1 enlists how these characteristics influence the service-oriented architecture and its application. Section 2.6.2 discusses the various logical entities in a sensor system that can be used as a service abstraction. Section 2.6.3 discusses the problems of Web service support within sensor networks. Finally, Section 2.6.4 discusses the state-of-the-art of service-orientation support within the WSN research domain.

2.6.1 Revisiting the service-orientation principles

A number of differentiating characteristics of sensor networks and systems need to be taken into account when developing sensor applications in a service-oriented manner [119, 122]. These are mainly related to the greater network dynamics, limited application scope, unreliable networking and available resources. More specifically:

- **Spread of resources.** While resources are scarce within a wireless sensor network, they are more abundant within the gateway and back-end tiers. This spread of resources needs to be leveraged to reduce the overhead caused by the application of service-orientation.
- **Simple(r) services.** While sensor systems can contribute to a wide variety of applications, the functionality that is therefor required within a sensor network has limited diversity [119]. It primarily entails sensing, actuation, and simple data processing. Consequentially, more simple interactions take place in sensor networks, which in turn reduces the complexity of service descriptions.
- **Scale and redundancy.** In contrast to traditional service-oriented systems, in which mostly a single or a few computing nodes provide a certain service, in sensor networks typically multiple nodes offer the same service. Diversification during service discovery should therefore be based on the context within which a service is provided [60]; i.e. physical context like location, but also hardware and software state; e.g. remaining energy, current memory use, available sensors, etc.
- **Increased platform heterogeneity.** Platform heterogeneity in sensor systems is more extreme than in the traditional distributed environments. Service-oriented solutions should operate comfortably on a wide range of underlying hardware and software platforms across the various sensor system tiers.
- **Unreliable networking.** Due to the application of low-power wireless networking and the mobility of sensor nodes, network connections in wireless sensor networks tend to be unreliable. Loose-coupling of services should not only be supported to improve reuse, but also to gracefully deal with intermittent connectivity.
- **Development-time vs run-time resources.** Service-orientation advocates for generic services that are build once and used in multiple applications. Its efforts are predominantly targeted at reducing development overhead, while run-time resources are second. In sensor

networks, the constrained resources increase the importance of reducing the run-time overhead, yet this may not lead to an impoverished development experience.

Finally, it is important to high-light a feature of service-orientation itself; i.e. it being a technology-neutral paradigm. As discussed in Section 2.5.2, it does not have to imply the use of Web services and, consequentially, verbose XML-based communication. More efficient and tailored approaches must be explored.

2.6.2 Service abstractions in sensor network research

An important diversifying factor among service-oriented platforms for sensor systems, is the logical entity that is considered as being a service. Similar to the multi-application approaches discussed in Section 2.1.1, these range from the entire *network*, over individual *nodes*, to software *modules*, and even individual *functions*. A conceptual representation is shown in Figure 2.5.

Entire sensor networks are for instance abstracted away by a Web service in the Oracle edge-server [136]. These Web services can then be used within multiple back-end service compositions, as shown in Figure 2.5a. At a slightly less coarse-grained level, individual sensor nodes can be represented as a service, as shown in Figure 2.5b. Atlas [89], for instance, makes use of OSGi [137] bundles that wrap entire sensor nodes and exposes them as device drivers on a gateway. While both these approaches enable the use of service-orientation in the back-end, they do not necessarily imply its application inside the sensor network. To provide such support, additional middleware support is required on sensor nodes that provide a run-time environment in which services can operate and be discovered. One approach is to present software modules as services, as shown in Figure 2.5c. This is for instance provided by µSMS [46] and USEME [22] that apply component models to create a service abstraction. Finally, individual functions provided by a middleware can be presented as services, as shown in Figure 2.5d. In TinySOA [152], for instance, middleware functions that perform sensing, actuation or control tasks are presented as services and invoked by queries that are distributed in the network.

For individual sensor nodes to be considered a reusable asset that serves multiple applications and users, the applied service concept might not abstract away that infrastructure. In other words, the basic granularity of a service needs to be smaller than that of a sensor node. This is either realised by using software modules as a service abstraction, or individual functions. The following sections therefore primarily deal with these forms of service abstractions.

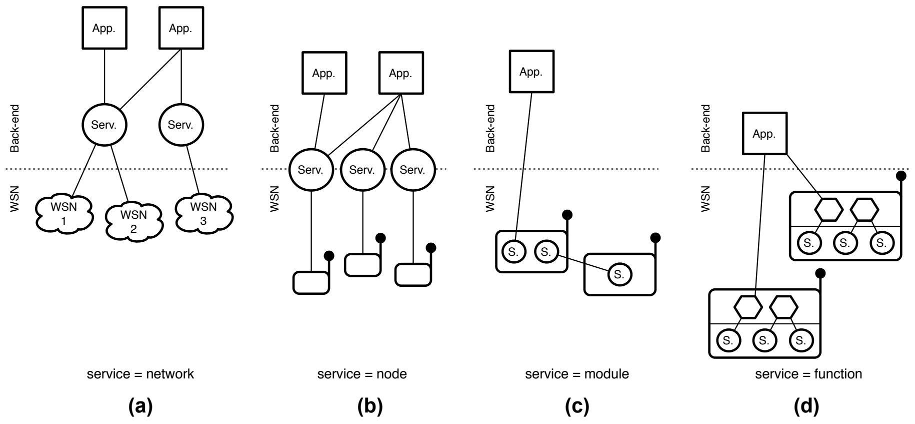


Figure 2.5 – The granularity of service abstractions in sensor systems ranges from entire sensor networks to individual functions.

2.6.3 A note on Web services and sensor networks

As discussed in Section 2.5.2, the most common implementation of the service-orientation paradigm is realised by the Web services stack, often referred to as WS-*. In terms of development support and integration with enterprise services, it would therefore make great sense to adopt WS-* in the sensor network as well. The WS-* stack was however not defined with resource-constraints in mind and its assumptions on available computational power, storage and bandwidth do not hold in constrained sensor networks [97, 101]. Successful efforts have however been made towards WS-* adoption, yet, at the cost of flexibility.

Tiny Web Services [148], for instance, (i) applies heavy optimisation of the TCP/IP stack (persistent TCP connections, disabling delayed acknowledgements, link layer retransmissions), (ii) supports only single-hop communication between sensor nodes and the gateway, (iii) requires additional gateway functionality in support of back-end/sensor node interactions, and (iv) only supports limited and optimised XML-parsing. While this shows the feasibility of WS-* adoption, it only offers a restricted mode-of-operation. One great disadvantage is that sensor nodes cannot function as Web service clients themselves, which eliminates in-network interactions. Furthermore, WSDL descriptions provided by the sensor nodes are still a few kilobytes in size. Together with the additional XML overhead during service invocations, the communication overhead remains large.

To make Web services more feasible on embedded devices the Devices Profile for

Web Services (DPWS) [130] was developed. DPWS supports a minimal subset of WS-* standards that includes secure sending of messages, service discovery, and eventing in support of machine-to-machine communication. While successfully applied on gateway devices as a proxy for constrained sensor nodes [30, 172], support for the latter themselves is limited. WS4D-uDPWS [190] provides an implementation in Contiki, yet supports only a basic subset of the DPWS functionality and features heavy compile-time optimisation by assuming the capabilities of a node to be static.

The most promising Web services approach within the WSN domain is probably by application of CoAP [156] together with OMA LWM2M [134] and IPSO SmartObject [79] resource descriptions. As discussed previously in Section 2.1.2, these are however only recent initiatives.

By consequence, numerous service-oriented platforms for sensor networks make use of proprietary service abstractions and communication. The following section provides an overview.

2.6.4 State-of-the-art service-orientation support

This section surveys a number of representational service-oriented application frameworks. These are evaluated along the same set of criteria as previously presented in Section 2.4.1. The sole exception is that the last column in Table 2.2, *OS heterogeneity*, is evaluated slightly differently; inter-tier interoperability is considered to be inherently provided in case a standardised service abstraction (e.g. WS-*, DPWS, etc.) is supported. This can either be realised on constrained nodes themselves, or by a translating proxy situated at the gateway or back-end. Only application frameworks that provide a service abstraction available within the sensor network tier are reviewed. This eliminates frameworks that provide gateway or back-end services that abstract away or wrap the sensor network tier.

TinySOA [152] applies a service-driven querying model, similar to the distributed databases approach. Queries are disseminated into the network and invoke services on targeted sensor nodes. Each service represents a single sensing, actuation or control function and returns a single scalar value. The set of services on a sensor node is statically defined and no modular configuration is provided at run-time. Behavioural configuration of services can be performed by means of their function parameters. Queries are specified in a event-condition-action format that additionally includes spatial scoping to determine target nodes. The event and condition statements allow to specify the context in which the service needs to be executed, and the action statement defines the service to invoke by means of a simple identifier. No structural representation of the

	Run-time software configuration					Run-time discovery				OS heterogeneity
	Structural configuration			Behavioural configuration	Targeting	Representation			Discovery scope	
	Modularity	Interactions				Semantic	Structure	Context		
		Scope	Configurable							
	o/m/s	o/l/d	o/i/e	o/i/e	nd/gr/nw	o/s/c	o/s/c	o/x	o/i/e	o/x/+
Requirements	**m**	**d**	**e**	**e**	**nd**	**c**	**c**	**x**	**e**	**+**
SOA platforms										
TinySOA [152]	o	l	i	**e**	gr	s	o	**x**	i	o
Servilla [51]	o	**l,d**	i	**e**	nw	s	s	o	i	o
WSN-SOA [101]	**s/m**	**l,d**	i	**e**	**nd**	s	?	o	**e**	**+**
µSMS [46]	**m**	**l,d**	i	o	**nd**	s	o	o	o	**+**
nSOM [47]	**m**	**l,d**	i	o	**nd**	**c**	**c**	o	**e**	**+**
OASiS [97]	o	**l,d**	i	o	gr	s	o	**x**	i	o
USEME [22]	?	**l,d**	i	i	gr	s	o	**x**	i	x
Dioptase [14]	s	**l,d**	i	o	**nd**	**c**	**c**	**x**	**e**	**+**

o/m/s: o = none, m = module, s = script - o/l/d: o = none, l = local, d = distributed - o/i/e: o = none, i = internal, e = external - nd/gr/nw: nd = node, gr = group, nw = network - o/s/c: o = none, s = simple, c = coordinated - o/x: o = no, x = yes - o/x/+: o = none, x = within a single tier, + = across tiers

Table 2.2 – Overview of the support provided by state-of-the-art service-oriented application frameworks towards the realisation of open and reusable sensor systems. (Criteria evaluations that fulfil the proposed requirements are in bold.)

service is available at run-time and an overview of the available services cannot be retrieved via service discovery. TinySOA is implemented only on TinyOS.

Servilla [51] applies a similar approach as TinySOA, but supports remote service invocations and provides a structural service interface representation. Applications are implemented by means of tasks that are disseminated over the entire network. Tasks are byte-code scripts that are interpreted by a virtual machine provided by each sensor node and invoke local or remote services. Services, in turn, are implemented by a compile-time defined set of modules. Platform-independent service representations enlist the name of a service, a list of functions with typed parameters and return values, and attributes that can serve behavioural configuration of the invoked service. Although Servilla's task abstraction hides away hardware heterogeneity, it is only implemented for TinyOS.

WSN-SOA [101] attempts to bridge the gap between high-end networked devices and constrained sensor networks by applying service-orientation principles. It integrates a light-weight proprietary protocol stack on constrained sensor nodes, with DPWS on more resource-rich sensor nodes and a WS-* stack on high-end nodes. On constrained sensor nodes, services are implemented as TinyOS components that expose parameters for behavioural configuration. A follow-up publication [99], however, reports that services can also be dynamically deployed by means of scripts. On more resource-rich platforms, services are implemented as dynamically deployable OSGi [137] bundles. Local and remote interactions between services are supported via both a messaging abstraction and a centralised topic-based publish/subscribe broker on a gateway. In cooperation with service discovery, this allows dynamic interactions between services, yet these are not externally controllable. While services provide a semantic representation of themselves, it is unclear whether and to which extend also a structural representation is provided.

µSMS [46] (micro-Subscription Management System) implements a service-oriented infrastructure based on a component model, software agents and publish/subscribe messaging. Services are implemented by component-based agents that are remotely deployable, but offer no support for additional behavioural configuration. Agents dynamically subscribe themselves to locally and remotely produced events of interest. Subscriptions are based on a simple numerical event type identifier, together with additional parameters such as publishing service identifiers and content-based filtering. Implementations of µSMS are available that operate on sensor nodes (TinyOS and Contiki), PDA's and laptops.

nSOM [47] (nano Service-Oriented Middleware) builds on µSMS and consequentially provides similar functionality. Its main contribution is the realisation of interoperability with Internet-based applications. While in-network interactions occur in a proprietary manner as in µSMS, services additionally provide RDF-compliant SMD/JSON service definitions that are translated to WSDL2.0 on dedicated broker nodes. This enables remote WS-* interactions and results in better support for the discovery of services. Implementations are realised for resource-rich Sun SPOT sensor nodes, Android smartphones and laptops. Interestingly, this implicitly confirms that XML/JSON based technologies are less suited for constrained sensor nodes, for which a prior µSMS implementation was realised.

OASiS [97] (Object-centric Ambient-aware Service-oriented Sensornet) applies an object-centric programming approach. It provides programming abstractions that allow a monitored phenomenon (called object) to drive application behaviour. Applications themselves take the form of a dynamic distributed service graph. Primarily targeted applications are for instance vehicle tracking and fire detection, in which over time the active service graph autonomously 'moves' throughout the network to follow the phenomenon. Nodes are pre-loaded with services, yet, these are dynamically activated and included in the service graph as the monitored phenomenon approaches the node. This dynamism is served by run-time discovery of services. Although services have well-defined interfaces consisting of typed input and output ports, such descriptions are compile-time structures only, and discovery occurs based on simple identifiers. Context parameters, such as location and energy level, are taken into consideration during the service discovery process. Interactions between services are dynamically established as the service graph evolves. Additionally, back-end based Web Services can be accessed and included in the service graph for greater computational resources. OASiS itself is however only implemented on TinyOS.

USEME [22] is a service-oriented and component-based framework that aims to facilitate WSN application development. It therefor combines macro-programming with node-centric programming. With the aid of a visual tool, a distributed service composition can be created that stores information of services, their interfaces and dependencies, and which is used to (partly) generate service code. It is unclear whether services can be individually deployed at run-time or if a comprising static image is flashed to nodes. Local and remote service dependencies do however get resolved at run-time to support node mobility. This is realised by means of service discovery based on service identifiers and additional context parameters. An API is provided to configure service parameters, yet this is not externalised and only available from within other services. USEME is implemented in Java and .NET, respectively for the high-end Sun SPOT and iMote2 sensor nodes.

Dioptase [14] implements a light-weight data stream management system for the Web-of-Things. It allows developers to describe complex fully-distributed stream-based mashups and to deploy them dynamically as task graphs in a sensor network. Tasks are represented as Web services and provide sensing, processing and actuating functionality. Tasks are either built into the middleware or specified in run-time deployable scripts using a domain-specific language. Task interactions are defined within the mashup and can be both local and remote. Dioptase relies on third-party service discovery to realise the mashups [60], yet an overview of streams and tasks is externally available in a RESTful manner. Each stream adheres to a schema that defines the semantics, structure and additional metadata (e.g. unit of measurement) of the exchanged data. Although the Dioptase middleware can be deployed on devices ranging from powerful sensor nodes to cloud servers, it is too resource-intensive for constrained sensor devices.

2.6.5 Conclusion

Service-orientation presents a promising approach toward realising open multi-purpose sensor systems. A common service abstraction across the sensor system tiers, combined with a platform-independent interaction mechanism, can abstract away the large heterogeneity in hardware and software. Additionally, loose-coupling of services can facilitate application logic reuse and aids in dealing with unreliable networking within sensor networks. Service descriptions and discovery further improve third-party (re)use of deployed application logic.

Yet, while good examples are at hand, the surveyed service-oriented frameworks do not provide adequate solutions to the challenges put forward within this dissertation. Features such as structural configuration and external configuration of interactions are not well supported. Instead, functionality and integration of services is often determined at development time. Consequentially, run-time available abstract descriptions are often simple and discovery of application logic is only supported for internal dependency resolution. This restricts the open use of the deployed services within various applications and reduces return-on-investment of the available infrastructure. To fully realise the open and multi-purpose use of sensor system infrastructure, various aspects of service-orientation thus need to be further dealt with. The contributions in the following chapters aim to do so.

2.7 Summary

This chapter overviewed a selection of the state-of-the-art in WSN research that is related to the specific problems dealt with in this dissertation; i.e. the challenges to the realisation of open multi-purpose sensor networks. First, WSN-specific solutions were presented that deal with multi-purpose sensor networks, dynamic software configuration, and run-time discovery of application logic. Second, the service-orientation paradigm was introduced as a promising software engineering approach towards open reusable software systems. Its most prominent features and characteristics were presented and reimagined within the scope of wireless sensor networks. Throughout the chapter, a selection of application frameworks were evaluated against the primary requirements that this dissertation puts forward for open multi-purpose sensor systems. While most of these frameworks share and acknowledge various requirements, none of them provide a high enough level of configurability in an open manner that allows for multiple users to manage various sensor applications on shared sensor network infrastructure. The following chapters present contributions that considerably improve the required support thereto.

Chapter 3

An application platform for open and multi-purpose sensor systems

This dissertation reports on three middleware contributions that together realise an application platform for open and multi-purpose sensor systems. This chapter shortly introduces these contributions and discusses at a high-level how they jointly deal with the challenges identified in Chapter 1. The subsequent chapters present the various contributions in more detail.

Figure 3.1 shows the three contributions of this dissertation; LooCI, SDlite, and TalkSens. They primarily contribute at the middleware level, yet also provide a suitable application programming abstraction. Furthermore, the node-level solutions presented in the figure are completed with additional back-end support that contributes to greater interoperability across the sensor system tiers. The following paragraphs introduce the various contributions and discuss how they jointly realise an application platform for open and multi-purpose sensor systems.

The ability to individually manage multiple applications on a single node is provided by LooCI's run-time configurable *component infrastructure*. LooCI allows for *application components* to be individually deployed and subsequently configured to suit evolving application requirements and local context. Application components expose event producing and consuming *interfaces* via which they exchange events over an *event bus* in a publish/subscribe-based manner. Within the work presented in this dissertation, these component

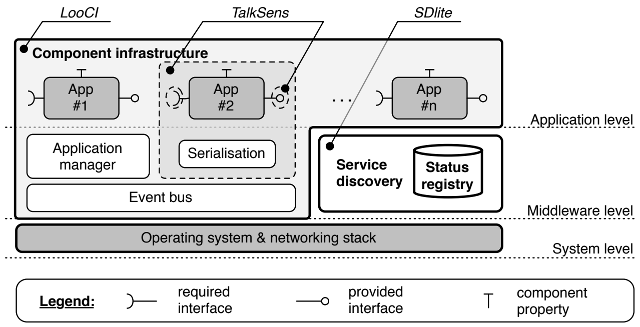


Figure 3.1 – Conceptual diagram highlighting this dissertation's middleware contributions towards more open and reusable sensor systems.

interfaces function as the provided service abstraction.[1] Interfaces are identified by the events they produce and consume, and as such are typed by event identifiers. Components additionally expose a set of configurable properties for behavioural configuration, and middleware support for configuration and inspection is provided by the *application manager*.

Status-aware service discovery, provided by SDlite, enables the discovery of LooCI components that provide a specified service under particular conditions. SDlite therefor features a *status registry* that provides generic sharing of status variables across software modules and layers on a sensor node. These can be system variables like remaining energy and number of deployed components, but also environmental variables such as position and temperature. To evaluate the status description included within service request, the service discovery mechanism operates in close relation with the status registry.

The last contribution, TalkSens, is *message definition framework* that enables multiple parties to come to a shared agreement of message types and contents, and provides associated development-time and run-time support. TalkSens provides a registry of platform-neutral event definitions that developers can use to select the events their LooCI components will exchange and generate specific serialisation code therefor. In addition, TalkSens enables run-time inspection of components for these event definitions.

[1]In this dissertation, the terms *application*, *component* and *service* are closely related when used in the context of the presented contributions. They are often used interchangeably, yet fine differences in meaning exist: *application* is used to denote a piece of application level logic; a *component*, or application component, is an individually managed unit-of-deployment that implements such application logic; and a *service* presents the external interfaces that components expose to their clients.

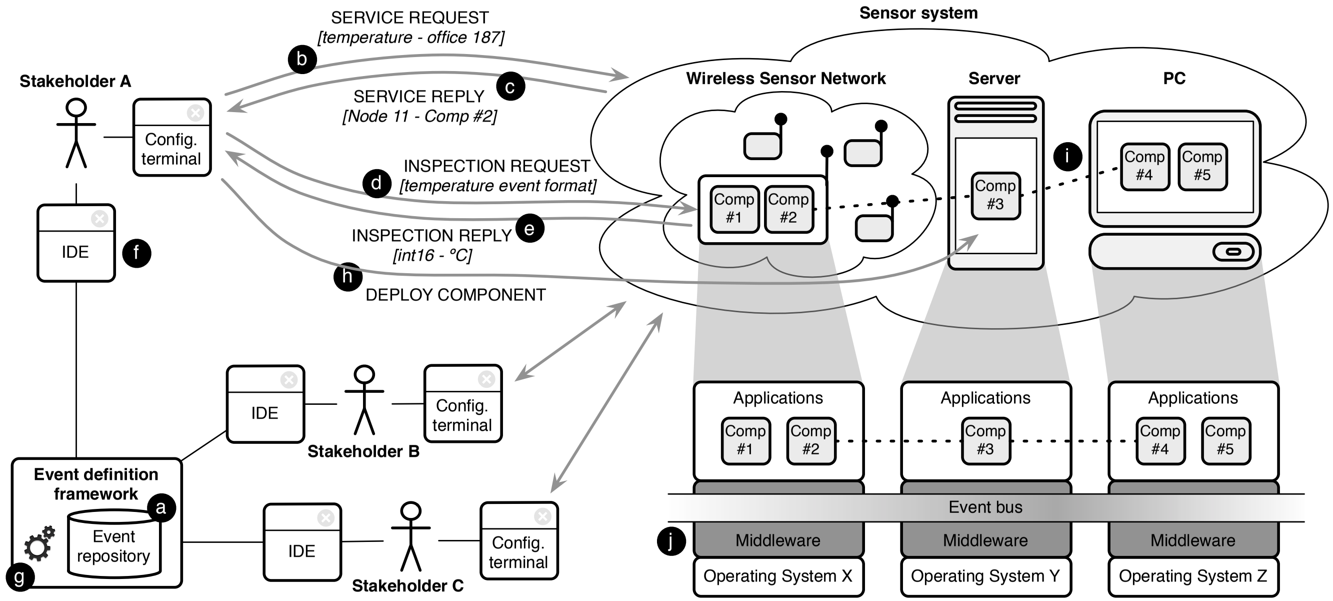


Figure 3.2 – Storyboard diagram illustrating the modus operandi provided by the integration of the various contributions presented in this dissertation.

The integration of the presented solutions effectively allows for a service-oriented modus operandi in which multiple stakeholders make use of a shared sensor system infrastructure. Figure 3.2 provides an exemplary storyboard diagram of how this works. Consider a stakeholder who wants to provide a client application with real-time temperature information at a certain location. To realise this, the stakeholder uses the event definition repository (*a*) to look up the temperature service identifier. Using this identifier, the stakeholder can discover whether such a service is already provided by a, possible third-party, sensor node deployed in the specified location (*b*). If that is the case (*c*), the provided service can be further inspected for details about the format in which temperature data is provided, i.e. the event definition (*d,e*). Using this information, a new back-end component can be developed (*f*), that includes generated event serialisation code (*g*), and is thus fully interoperable with the third-party temperature sensing service. Upon completion, this component can be deployed (*h*) and configured to receive temperature data and forward it to the client application (*i*). Such multi-tiered integration of application logic is enabled by the common middleware (*j*) that is deployed on the heterogeneous set of devices that constitute the sensor system. The result is a single, unified manner of application development and management that is applied across the entire sensor system. This enables multiple stakeholders to easily deploy application components on shared infrastructure, and make them interact with each other.

The following chapters present the various contributions in more detail. Chapter 4 presents LooCI, Chapter 5 presents SDlite, and Chapter 6 presents TalkSens.

Chapter 4

Run-time reconfigurable and modular distributed applications

This chapter presents the first contribution of this dissertation; the LooCI[1] component infrastructure. LooCI provides light-weight abstractions for application-level modularity and distributed interactions, and supports run-time reconfiguration of both. This breaks with the largely static and often homogeneous implementations of distributed sensor system applications, and enables more application dynamism, platform heterogeneity and reuse of application logic in such systems.

LooCI has been designed and implemented in collaboration with other members of our research group. Personal contributions mainly focus on component interaction and include the implementation for the Sun SPOT platform. This chapter presents the second version of LooCI, which adds a number of features to the previous version. These include multi-instantiation of components, component properties, and unified event-based management across all implementations.

Within this chapter, Section 4.1 lists detailed requirements for application-level modularity and distributed interactions within open multi-purpose sensor systems. Section 4.2 explores the features of component-based software engineering and the publish/subscribe communication paradigm, and discusses

[1] pronounced 'Lucy'

why these are well suited to fulfil said requirements. Section 4.3 introduces LooCI itself, and Section 4.4 discusses its various implementations on a range of software platforms.

4.1 Requirements

Using a sensor system deployment as an open and reusable infrastructure, requires support for the independent run-time management of individual application modules and the ability to create distributed applications with these modules as building blocks. More specifically, modularity and communication abstractions are needed that adhere to the following requirements:

1. **Application-level modularity.** An abstraction is needed that encapsulates independent units of application logic. Individual deployment at run-time and concurrent operation of such units on a single (sensor) node is essential.

2. **Flexible distributed communication.** A single communication abstraction is required that facilitates interactions between application units, both locally and remotely across all tiers in a sensor system. Common interaction patterns, such as one-to-one, one-to-many and many-to-one, need to be supported.

3. **Loose-coupling.** Relationships between application modules must be loose-coupled to enable flexible reuse of those modules, and gracefully deal with intermittent network connectivity and unpredictable behaviour of third-party application logic.

4. **Extensive reconfiguration and inspection at run-time.** Run-time reconfiguration and inspection of application modules, their behaviour, and their relationships is needed to enable post-installation and third-party use of sensor system infrastructure.

5. **Platform and language independence.** Both the modularisation and communication abstractions need to be platform- and language-independent to facilitate building distributed applications that operate on heterogeneous platforms across the sensor system tiers.

6. **Applicable on resource-constrained platforms.** Implementations of the presented solution must be feasible on Class 1 resource constrained devices (~100kB Flash, ~10kB RAM) [16].

4.2 A component- and event-based approach

To comply with the enlisted requirements, LooCI applies two well-known software engineering solutions: *component-based software engineering* (CBSE) and the *publish/subscribe* (pub/sub) communication paradigm. Components serve well as a modularity abstraction, while publish/subscribe-based communication suits the posed communication requirements. Their combined application is however extra powerful in terms of loose-coupling and reconfigurability. The following two sections introduce both technologies and discuss their applicability to meet the presented requirements.

4.2.1 Component-based software engineering

Component-based software engineering applies the principle of *separation of concerns* to building software solutions. It promotes decomposing larger problems into smaller sub-problems and solving the latter independently by implementing a solution in a self-contained logical unit, called a *component*. By composing together a set of such loosely-coupled components, a solution to the original larger problem can be built. When designed in a generic manner, individual components can furthermore be (re)used within multiple compositions. Both independent development of simpler components, and their reuse are expected to reduce overall development time [141].

The widely-accepted definition of a component, presented by Szyperski [165], states:

> *"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 can thus be regarded as an encapsulated unit of application logic that enables developers to build applications at a higher level of abstraction; i.e. by composing various, possible components together. While at times restricted to development-time, in many cases composition is a run-time operation. Explicit interface and dependency specifications, available either as meta-data or via component inspection, thereby allow third-party use of deployed components. These features satisfy to a large extent requirements one and four as stated in Section 4.1.

Specifically in the context of sensor networks, component models have been applied as a means to move away from early and impractical one-size-fits-all approaches [125]. This includes run-time evolution of software [32], both via fine-grained reprogramming of sensor nodes [167, 127], and via self-adaptation [70]. Yet, component models have also been adopted to increase ease-of-programming [31], provide language-independence [25] and share application logic [34].

Existing component models targeted at sensor networks, however, lack in support of the posed requirements. NesC [56], which is used to implement TinyOS [106], only provides a component model at development-time. During compilation, component boundaries are removed to build a highly optimised monolithic block of executable code. This prevents later run-time deployment of individual application components. Runes [26, 25], FiGaRo [127], Lorien [145] and RemoWare [167], are examples of run-time configurable component models. Yet, they do not provide explicit, reconfigurable abstractions for distributed interactions. While such interactions can be hard-coded or implemented in dedicated components, this limits the extent to which distributed relationships can be configured. Furthermore, configuration of local relationships is often limited to automated dependency resolving [127, 145, 167], which limits the freedom of configuration. Finally, with Runes being a notable exception, most WSN component models are implemented only for specific WSN platforms, and are not available for mobile and back-end devices (see Table 2.1, page 30). This limits their applicability in multi-tier sensor systems.

4.2.2 The publish/subscribe communication paradigm

The publish/subscribe communication paradigm [44] is a loosely-coupled form of event-based interaction that is often applied in large-scale and widely distributed environments. It provides indirect communication between *publishers* and *subscribers*. Subscribers express their interest in a certain event at an intermediate entity, at times referred to as an *event service*. Subsequently, when a publisher publishes an event to the event service, all subscribers with matching interests are notified thereof. The strength of the publish/subscribe paradigm lies in the flexibility that comes with the full decoupling that it realises between publishers and subscribers.

Decoupling is primarily the result of the indirection introduced by the event service and takes place along three dimensions. First, interacting parties are decoupled in *space* as they do not hold any reference to each other. Such information is only held at the event service, resulting in interacting parties that are unaware of each other. Second, interacting parties are decoupled in *time*. Both need not actively participate in the interaction at the same time as the event service can serve as a buffer. Events can be published at a time that a subscriber is disconnected, yet delivered when the latter reconnects. Third, *synchronisation* decoupling is realised by asynchronous communication between interacting parties. Publishers are not blocked during event publication until all subscribers are notified, and subscribers can get notified asynchronously about any new event while performing some other activity.

The publish/subscribe paradigm allows to build distributed applications that are less rigid and static in nature than when more traditional one-to-one synchronous interaction styles are used. The extra indirection allows to establish late bindings between publishers and subscribers, which can thus come and go on an individual basis. Additionally, decoupling allows to gracefully deal with intermittent network connectivity and unpredictable third-party behaviour. While such situations may still result in events not being published or delivered, this does not further impact the operation of an interacting partner. Finally, the publish/subscribe paradigm inherently provides a many-to-many communication model. These features correspond well with requirements two, three and four, as stated in Section 4.1.

The publish/subscribe paradigm has been widely applied in sensor networks as it fits well with their data collection and event-based nature. Additionally, loose-coupling aids in dealing with the wide distribution and lossy network connectivity. TinyCOPS [66] and Mires [161], for example, provide dedicated publish/subscribe implementations for sensor networks. As discussed in Section 2.6.4, WSN-SOA [101], µSMS [46] and nSOM [47] apply it as the communication mechanism of choice in their service-oriented frameworks. Also more industry-minded standardisation efforts like MQTT-SN [162] acknowledge the strengths of this interaction style. What is largely missing in the current state-of-the-art is the combination of support for (i) run-time configurable subscriptions, and (ii) an integration with a run-time configurable component model. For example, TinyCOPS and Mires only support the former, while WSN-SOA, µSMS and nSOM only support the latter.

4.3 The LooCI component infrastructure

LooCI, or the Loosely-coupled Component Infrastructure, is a platform-independent component infrastructure that defines an application-level component model and supporting middleware for configuration and communication purposes. An overview of LooCI's node-level architecture is shown in Figure 4.1. A pre-deployed middleware layer enables run-time deployment of application components and further configuration and inspection of their properties and relationships. Communication between components is provided via a fully distributed version of the publish/subscribe communication paradigm, which

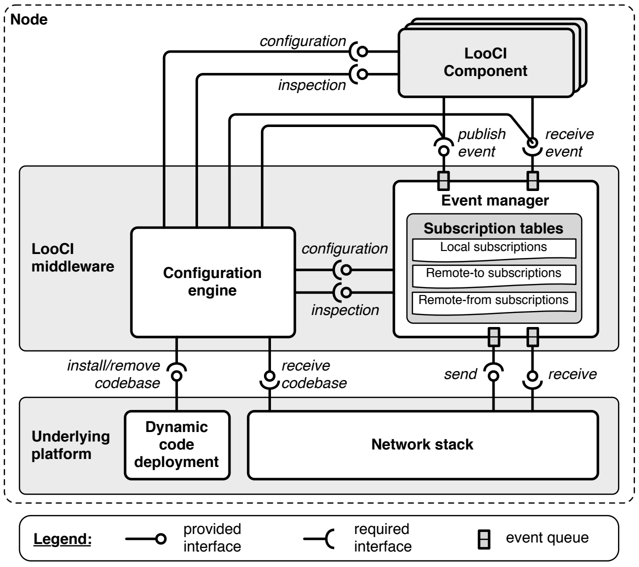


Figure 4.1 – Architectural diagram of the LooCI component infrastructure.

is implemented by the *event manager*. Configuration and inspection of both components and middleware is supported by means of the *configuration engine*. LooCI is defined independently from underlying platforms, yet relies on the network stack and dynamic code deployment provided by these platforms.

The following sections highlight the various features of LooCI and describe how these meet the requirements posed in Section 4.1. Full technical details are available at the LooCI website [77].

4.3.1 An application-level component model

The LooCI component model provides an application-level abstraction for building distributed applications within a sensor system. A LooCI component, as depicted in Figure 4.2, encapsulates application logic into an individually manageable entity. Components interact with each other only by exchanging events; they receive events via their *required interfaces* and publish events via their *provided interfaces*. Interfaces are typed and can only receive or publish events of the same type (see Section 4.3.2). Required interfaces thus explicitly specify a component's dependencies on the types of events it consumes, while provided interfaces explicitly define which type of events it produces.

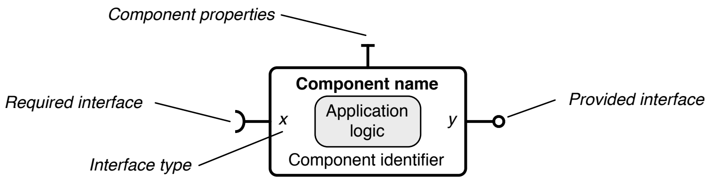


Figure 4.2 – Conceptual diagram of a LooCI component.

Components can have zero or more required/provided interfaces. The behaviour of a component is implemented in its *application logic*, which is configurable at run-time via *component properties*. While the *component name* provides a descriptive identifier, each component is uniquely identified by the combination of the locally unique numerical *component identifier*, and the network address of the node on which it executes.

LooCI components expose two management interfaces to the underlying middleware; i.e. a *configuration interface* and an *inspection interface* (shown in Figure 4.1). Both are locally accessible by the configuration engine, which in turn can be remotely accessed via the event bus. Via the configuration interface, components can be (de)activated and have their properties updated. The inspection interface enables inspection of a component's identifier, its state, provided and required interfaces, and properties. More details on component configuration and inspection are presented in Section 4.3.3.

Within LooCI, a component is both a unit-of-development as well as a unit-of-execution. Compiling a component results into a *codebase*, which functions as LooCI's unit-of-deployment.[2] Figure 4.3 depicts a LooCI component's life cycle. After deployment, codebases can be *instantiated* one or more time(s), with each new instance being a component of the codebase's type. Such multiple

[2]As explained in the text, a clear distinction exists between the notion of a *codebase* and a *component*. However, while a codebase is the unit-of-deployment, this text often uses the phrase '*deploy a component*' for brevity reasons. This implies that a codebase is deployed that implements that component.

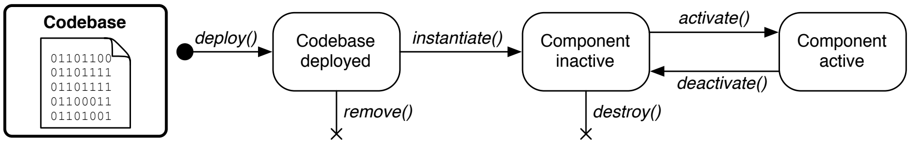


Figure 4.3 – State diagram of the LooCI component life cycle.

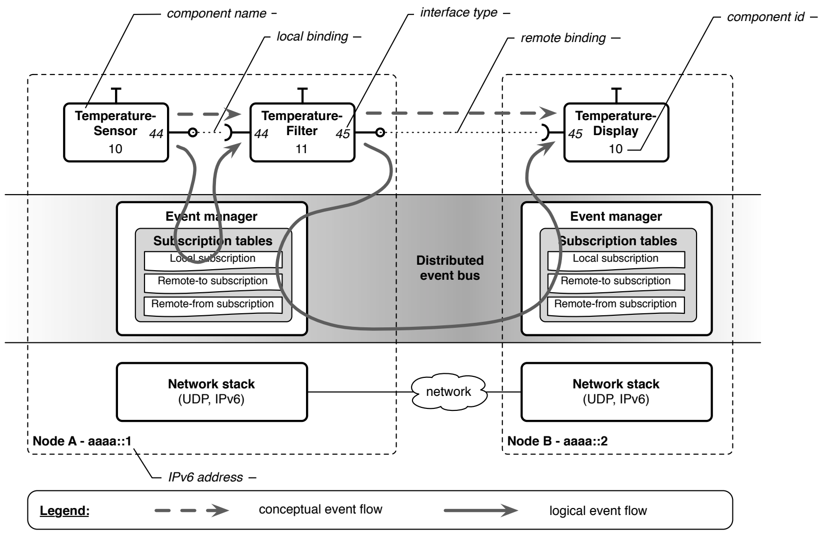


Figure 4.4 – The LooCI distributed event bus.

instantiation allows for codebases to be deployed once, yet instantiated multiple times with different behaviour based on their property configuration. This greatly reduces expensive over-the-air deployment, mostly of generic components (e.g. data filter). Upon instantiation, a component identifier is assigned a node-local value and the component holds an *inactive* state. Inactive components do not execute their logic and cannot produce or consume events, but can be further configured and inspected. After *activation*, a component becomes fully operative; they execute their logic, can produce and consume events and remain configurable and inspectable. Active components can be deactivated again and inactive components can be *destroyed*. This results in the removal of their run-time instance and all associated data from memory. In case all component instances of a codebase are destroyed, the codebase itself can be removed from the node's memory.

4.3.2 Loose-coupling: the distributed event bus

As briefly mentioned in the previous section, LooCI components only interact with each other via the event bus. This bus provides a fully distributed version of the publish/subscribe communication paradigm. A graphical presentation is provided in Figure 4.4. Each LooCI node implements its own local part of the bus within its *event manager*. Event managers administer only locally

```
0                   1                   2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...+-+-+
|        event type id          |  src comp id  |    payload
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...+-+-+
```

Figure 4.5 – The LooCI event format.

relevant event subscriptions and perform event dispatching towards local and remote components based on these descriptions. This decentralised approach allows for direct exchange of events between interacting nodes, which reduces network traffic and results in faster and more efficient event dispatching [44]. By publishing events to the event manager, components place events on the distributed event bus; conversely, components receive subscribed-to events from the event manager (see Figure 4.1).

Component interaction can thus be described on three levels of abstraction, of which two are depicted in Figure 4.4. Conceptually, components exchange events directly with each other via binding of their interfaces. Logically, this is implemented by means of the distributed event bus, which provides loose-coupling between the communicating components. And physically, exchanged events are encapsulated in network packets and routed over whichever network is provided between the communicating nodes. The latter is omitted from the figure for the sake of clarity.

Events and their types

LooCI provides a topic-based version of the publish/subscribe paradigm, in which subscriptions occur based on the subject, or topic, of events [44]. Within LooCI, these topics are identified by event types. In contrast to most other publish/subscribe systems, event types are not only represented by a descriptive human-readable name, but also by a more compact numeric identifier. The latter are sequentially assigned to the set of event types, resulting in a very compact format. During event exchange, only the numeric identifiers are used, which reduces communication and storage overhead. Real-world deployments [34, 72, 168], however, later showed that such naive numbering leads to extensive configuration overhead. Chapter 6, therefore presents a novel event typing system that incorporates more rich semantic information in the event type identifiers, with a limited size overhead.

LooCI events are formatted as shown in Figure 4.5. A small header consists out of a two-byte event type identifier followed by a single byte source component identifier. These provide the event manager with enough details to perform event dispatching, where needed together with a source node address provided by the underlying network stack. After the header, the event's payload consists out of an arbitrary array of bytes that represent data related to the event's type.

Event dispatching

Dispatching of events to local and/or remote components is based on the entries stored in *subscription tables* (see Figure 4.4). Each event manager implements three such tables:

- **Local subscription table:** used to dispatch events between two local components based on entries in the following format:

  (event type, source component ID) → (destination component ID)

- **Remote-to subscription table:** used to dispatch locally published events to remote nodes based on entries in the following format:

  (event type, source component ID) → (destination node address)

- **Remote-from subscription table:** used to dispatch incoming remote events to locally subscribed components based on entries in the following format:

  (event type, source component ID, source node address) → (destination component ID)

When events enter the event manager, either from publishing components or from the network, the event manager compares the incoming event with the entries in the respective tables. For any matching entry, the incoming event is copied and the copy dispatched according to the subscription entry.

Based on subscriptions, bindings between two components can be created, as shown in Figure 4.4. A *local binding* between components on the same node only requires an entry in the local subscription table. A *remote binding* between components on different nodes requires entries in both remote subscription tables. On the event producing side of the binding, an entry in the remote-to subscription table forwards any matching locally produced event to the specified destination node. At the event consuming side, an entry in the remote-from subscription table dispatches any matching events received over the network to the locally subscribed component.

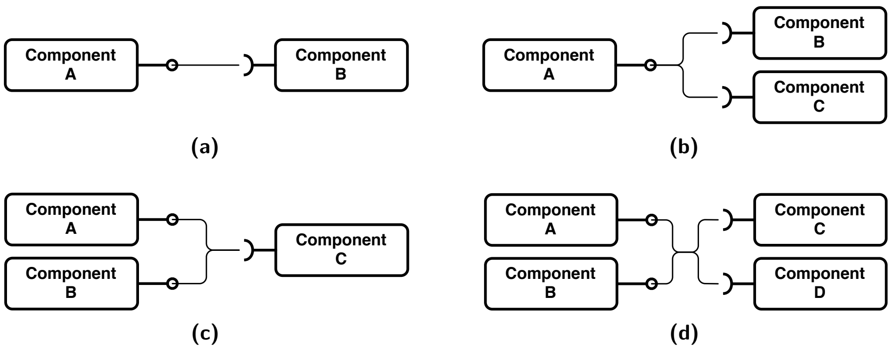


Figure 4.6 – LooCI's binding model allows for a wide range of interaction modalities; *(a)* one-to-one, *(b)* one-to-many, *(c)* many-to-one, and *(d)* many-to-many.

LooCI's binding model allows for various interaction modalities; one-to-one, one-to-many, many-to-one, and many-to-many, as shown in Figure 4.6. These are further facilitated by means of *wildcards* that can be used instead of specific values for event types, component identifiers and node addresses. Such use of wildcards facilitates (i) more rich event dispatching and (ii) opportunistic bindings. For instance, the event type wildcard enables forwarding of all locally published events to a logger-component with a single subscription, while the node address wildcard enables broadcasting of events to the dynamic set of nodes within range.

The use of subscription tables to store binding information, effectively decouples components. Components merely publish and receive events and are unaware of what happens during event dispatching. As components do not store any information concerning their interaction partners, components are decoupled in space. With regards to time decoupling, LooCI does not persist events that cannot be delivered instantaneously. Instead, LooCI provides a best-effort service during event dispatching; if a subscriber cannot be reached, the event is dropped and normal operation continues. This strategy is most often permitted as in sensor network applications a loss of (a few) events in a monitoring stream is considered tolerable. Finally, synchronisation decoupling is achieved by the use of event queues. Upon publication, a component places its event in a queue towards the event manager, which allows the latter to process the event in an asynchronous manner and prevents the component from blocking during event dispatching. Additionally, queues toward the network and event receiving components prevent the event manager from blocking on long-lasting component and network operations.

In support of those cases where a synchronous request/reply-style communication is needed, LooCI provides *directed events*. These are for instance used to target management events to a specific component or node. Upon event publication, the source component specifies the targeted destination component and node. This makes the event bypass the subscription tables and be directly sent to its destination component. Usage of directed events is however discouraged for application components, as it breaks loose-coupling of components.

Finally, the event manager provides configuration and inspection interfaces to remotely manage component bindings, or event subscriptions (shown in Figure 4.1). These interfaces respectively allow to create and remove subscriptions, and get an overview of existing subscriptions. More details on configuration and inspection are presented in the following Section.

4.3.3 Run-time reconfiguration and inspection

Management of a LooCI node occurs via the *configuration engine*, which is part of the LooCI middleware, as shown in Figure 4.1. It implements two management APIs, the *configuration API* and the *inspection API*, that can be interacted with both locally and remotely. Local interaction occurs by means of direct method invocation from within other middleware modules or application components. Remote interaction occurs by means of events. Therefor, a set of predefined configuration and inspection events are defined that invoke the respective configuration and inspection methods upon event reception. Besides providing these management interfaces, the configuration engine stores a list of all deployed codebases, and maintains references to the configuration and inspection interfaces of all instantiated components and the event manager.

The configuration API, shown in Listing 4.1 allows for structural and behavioural configuration of the application logic (i.e. components) on a node, as well as configuration of application-level relationships (i.e. subscriptions). This API allows for the deployment and removal of codebases (lines 2 - 3), instantiation of codebases and destruction of components (lines 4 - 5), (de)activation of components (lines 6 - 7), the configuration of property values (line 10), and the creation and removal of subscriptions (lines 13 - 19).

A close look at the API reveals that parameters related to the configuration commands are primarily based upon the numerical identifiers of event types, codebases, components and properties. This elimination of human-readable string based identifiers (i.e. names) significantly reduces the payload sizes of configuration and inspection events. This does however give rise to a number of additional inspection commands that allow to retrieve a more human-readable description of codebases, components and properties based on their numerical

```
/* Component lifecycle configuration */
(ResultCode, CodebaseID) deployCodebase(NodeAddress node, String
     codebaseName)
ResultCode removeCodebase(NodeAddress node, uint8 codebaseID)
(ResultCode, ComponentID) instantiateCodebase(NodeAddress node, uint8
     codebaseID)
ResultCode destroyComponent(NodeAddress node, uint8 componentID)
ResultCode activateComponent(NodeAddress node, uint8 componentID)
ResultCode deactivateComponent(NodeAddress node, uint8 componentID)

/* Property configuration */
ResultCode setProperty(NodeAddress node, uint8 componentID, uint16
     propertyID, uint8 size, uint8[] value)

/* Binding configuration */
ResultCode wireLocal(NodeAddress node, uint16 eventType, uint8
     srcComponentID, uint8 dstComponentID)
ResultCode wireRemoteTo(NodeAddress node, uint16 eventType, uint8
     srcComponentID, NodeAddress dstNode)
ResultCode wireRemoteFrom(NodeAddress node, uint16 eventType, NodeAddress
     srcNode, uint8 srcComponentID, uint8 dstComponentID)
ResultCode resetWires(NodeAddress node, uint8 componentID)
ResultCode unwireLocal(NodeAddress node, uint16 eventType, uint8
     srcComponentID, uint8 dstComponentID)
ResultCode unwireRemoteTo(NodeAddress node, uint16 eventType, uint8
     srcComponentID, NodeAddress dstNode)
ResultCode unwireRemoteFrom(NodeAddress node, uint16 eventType, NodeAddress
      srcNode, uint8 srcComponentID, uint8 dstComponentID)
```

Listing 4.1 – The LooCI configuration API.

identifiers. Furthermore, each configuration (and inspection) command returns a result code. This code informs the application manager whether the command was successful, or in the other case, the cause of error (e.g. codebase not found, wire already exists, insufficient memory).

The inspection API, shown in Listing 4.2, can be used to retrieve information about the current configuration of a node. It provides access to run-time information on deployed codebases and their component instances; i.e. a list of all deployed codebases (line 2) and their ID or name (line 3 - 4), all components that are instances of a certain codebase (line 5) and their matching codebase name or ID (line 6 - 7), the set of all components (line 8), and the state of a component (line 9). Component properties can also be inspected by means of commands that retrieve all property IDs of a component (line 12), the configured

```
/* Codebase and component inspection */
(ResultCode, Collection<CodebaseID>) getAllCodebaseIDs(NodeAddress node)
(ResultCode, CodebaseName) getCodebaseNameByID(NodeAddress node, uint8
     codebaseID)
(ResultCode, Collection<CodebaseID>) getCodebaseIDsByName(NodeAddress node,
      String codebaseName)
(ResultCode, Collection<ComponentID>) getComponentIDsByCodebaseID(
     NodeAddress node, uint8 codebaseID)
(ResultCode, CodebaseID) getCodebaseIDByComponentID(NodeAddress node, uint8
      componentID)
(ResultCode, CodebaseName) getCodebaseNameByComponentID(NodeAddress node,
     uint8 componentID)
(ResultCode, Collection<ComponentID>) getAllComponentIDs(NodeAddress node)
(ResultCode, ComponentState) getComponentState(NodeAddress node, uint8
     componentID)

/* Property inspection */
(ResultCode, Collection<PropertyID>) getPropertyIDs(NodeAddress node, uint8
      componentID)
(ResultCode, PropertyValue) getPropertyValue(NodeAddress node, uint8
     componentID, uint16 propertyID)
(ResultCode, PropertyType, PropertyName) getPropertyInfo(NodeAddress node,
     uint8 componentID, uint16 propertyID)

/* Interface inspection */
(ResultCode, Collection<EventType>) getRequiredInterfaces(NodeAddress node,
      uint8 componentID)
(ResultCode, Collection<EventType>) getProvidedInterfaces(NodeAddress node,
      uint8 componentID)

/* Binding inspection */
(ResultCode, Collection<EventType, ComponentID, ComponentID>) getLocalWires
     (NodeAddress node, uint16 eventType, uint8 srcComponentID, uint8
     dstComponentID)
(ResultCode, Collection<EventType, ComponentID, NodeAddress>)
     getRemoteToWires(NodeAddress node, uint16 eventType, uint8
     srcComponentID, NodeAddress dstNode)
(ResultCode, Collection<EventType, ComponentID, NodeAddress, ComponentID>)
     getRemoteFromWires(NodeAddress node, uint16 eventType, uint8
     srcComponentID, NodeAddress srcNode, uint8 dstComponentID)

/* Platform inspection */
(ResultCode, PlatformType) getPlatformType(NodeAddress node)
```

Listing 4.2 – The LooCI inspection API.

value of a property (line 13), and a full property description (line 14), which includes the data type and name of the property. Furthermore, a component can be inspected for its required (line 17) and provided (line 18) interfaces, while the event manager can be inspected for the configured local (line 21) and remote (lines 22 - 23) subscriptions. Finally, a node can be inspected for its platform type (line 26), which specifies the underlying software platform.

Most commonly, the management APIs are interacted with remotely. LooCI therefor provides the *management client*; a shell-like configuration tool that supports a set of commands to disseminate configuration and inspection events into the sensor network. This tool is further demonstrated in the following section.

4.3.4 LooCI configuration in practice

This section presents the practicalities of configuring a distributed sensing application with LooCI. The temperature monitoring application depicted in Figure 4.4 is used as a practical example. The basic composition consists of a *TemperatureSensor* component and *TemperatureFilter* component on a constrained sensor node (i.e. `node_A`), and a *TemperatureDisplay* component on a more resource-rich back-end node (i.e. `node_B`). The behaviour of these components adhere to the following:

TemperatureSensor. This component measures the environmental temperature every 10 seconds, wraps this data in a `raw_temperature_event`, and publishes these events on the event bus. It provides a `sample_frequency` property via which the measurement frequency can be configured, and has one provided interface of type `raw_temperature_event`.

TemperatureFilter. This component filters out temperature readings that only differ marginally from prior readings, and thus limits the number of events that need to be transmitted to the back-end. It provides two properties; a `temperature_delta` which specifies the minimum delta that needs to be exceeded in successive temperature measurements to forward a new measurement, and a `refresh_rate` that specifies the minimum frequency at which a new temperature event should be forwarded. The component has one required interface of type `raw_temperature_event`, and one provided interface of type `filtered_temperature_event`.

TemperatureDisplay. This component displays the temperature information it receives. It has a single required interface of type `filtered_temperature_event`.

```
Welcome to the LooCI management client.
Type 'help' if you need any.
>> getComponentIDs node_A
[1,10]
>> getCodebaseNameOfComponent 10 node_A
TemperatureSensor
>> getState 10 node_A
1
>> getProperties 10 node_A
[1]
>> getPropertyInfo 1 10 node_A
type: short, name: sample_frequency
>> getProperty 1 10 node_A short
10
>> deploy temperature_filter.comp node_A zigduino
11
>> deploy temperature_display.jar node_B osgi
10
>> instantiate 11 node_A
11
>> instantiate 10 node_B
10
>> getProvidedInterfaces 10 node_A
[raw_temperature_event]
>> wireLocal raw_temperature_event 10 11 node_A
success
>> wireTo filtered_temperature_event 11 node_A node_B
success
>> wireFrom filtered_temperature_event 11 node_A 12 node_B
success
>> getProperties 11 node_A
[1,2]
>> getPropertyInfo 1 11 node_A
type: byte, name: temperature_delta
>> setProperty 1 11 node_A 1 byte
success
>> getPropertyInfo 2 11 node_A
type: byte, name: refresh_rate
>> setProperty 2 11 node_A 30 byte
success
>> activate 11 node_A
success
>> activate 12 node_B
success
```

Listing 4.3 – A print-out of the LooCI management client during configuration of the temperature monitoring application.

A precondition to the configuration of the presented composition is that a third-party has already deployed, instantiated and activated the TemperatureSensor component on `node_A`.

Listing 4.3 provides a print-out of the configuration actions that are performed using the LooCI management client. First, `node_A` is inspected to determine which components are instantiated (lines 3 - 6) and whether the *TemperatureSensor* component is active (lines 7 - 8).[3] Once an active *TemperatureSensor* component is detected, it is inspected for its properties, more precisely the currently configured `sample_frequency` value. This current value corresponds with the configuration requirements, and the component can thus be reused. This reuse is independent from any other compositions of which this component is a part. Configuration continues with the deployment of the *TemperatureFilter* codebase on `node_A` and the *TemperatureDisplay* codebase on `node_B` (lines 15 - 18). This results in their respective codebase identifiers being returned. Using these codebase identifiers, the codebases can be instantiated, which returns the respective component identifiers (lines 19 - 22). To be able to correctly configure the appropriate component bindings, first the provided interfaces of the *TemperatureSensor* component are inspected (lines 23 - 24). As there is only one, which is of the expected type, the components can be wired.[4] A local binding between the *TemperatureSensor* and *TemperatureFilter* components is established (lines 25 - 26), as well as a remote binding between the *TemperatureFilter* and the *TemperatureDisplay* components (lines 27 - 30). Then, the *TemperatureFilter* component is inspected for its properties (lines 31 - 32). Two property identifiers are returned and both properties are further inspected for additional information to allow their correct configuration (lines 33 - 40). The `temperature_delta` is set to 1° Celsius, and the `refresh_rate` to 30 to make sure that new temperature data is received at the *TemperatureDisplay* component at least every 5 minutes, or when the temperature has changed more than 1° Celsius since the last reported value. Finally, both the *TemperatureFilter* and *TemperatureDisplay* components are activated at the respective nodes, after which temperature data will be displayed at the back-end node as required.

Although the LooCI management client provides an application manager with great control over a sensor system, its low-level interface can quickly render sensor system management into a complex task. The small configuration example discussed here, for instance, already requires more than 20 management commands to complete. More high-level management abstractions are a natural

[3]Component identifier *1* is reserved within LooCI for the configuration engine. User components are numbered starting from *10*.

[4]For clarity reasons, the LooCI management client can be configured to resolve event type identifiers to human-readable names. This is strictly a back-end process; under the hood, event type identifiers are used during configuration and inspection.

	Sensor network tier		Gateway tier	Back-end tier
Software platform	**Contiki**	**Sun SPOT**	**OSGi**	
Programming language	ANSI C	Java ME CLDC1.1	Java SE	
Hardware platform	Zigduino/ AVR Raven	Sun SPOT	e.g. RaspBerry Pi B+	e.g. Dell OptiPlex 780
Processor speed	16 MHz/8 MHz(#)	180 MHz	700 MHz	2,83 GHz
Storage (Flash)	128 kB/128 kB	4 MB	~ MB - GB	250 GB
Memory (RAM)	16 kB/16 kB	512 kB	512 MB	8 GB

(#) The AVR Raven has a theoretical maximum clock speed of 20 MHz, but is clocked to 8 MHz by default.

Table 4.1 – An overview of the computational and memory resources that are provided by the platforms for which LooCI has been implemented.

complement and have consequentially been developed within our research group [35, 69].

4.4 LooCI ports across the sensor system tiers

Easy development and configuration of multi-tier sensor applications demands a unified approach thereto across the various tiers. LooCI has therefore been implemented on three well-selected software platforms that operate on a range of hardware devices. Table 4.1 provides an overview of the hardware and software platforms for which LooCI was implemented. The three LooCI ports are:

LooCI/Contiki. The Contiki [38] port provides an ANSI C implementation of LooCI. It enables support for constrained sensor nodes like the Zigduino[5] or AVR Raven[6]. Both operate at a processor speed in the lower MHz range and have limited amounts of storage and memory, respectively 128kB and 16kB.

LooCI/SunSPOT. The Sun SPOT [9] port provides a Java ME CLDC1.1 implementation, which supports a more resource-rich type of sensor nodes. The Sun SPOT platform comes with dedicated hardware that features processing speed, storage and memory at an order of magnitude larger than the typical Contiki-based sensor nodes.

LooCI/OSGi. The OSGi [137] port provides a standard Java SE implementation and brings LooCI support to both the gateway and the back-end tier. Gateway devices, such as the RaspBerry Pi[7], operate at processor speeds

[5]`http://www.logos-electro.com/zigduino/`

[6]`http://www.atmel.com/tools/avrraven.aspx`

[7]`https://www.raspberrypi.org/`

around the 1Ghz frontier, and their memory and storage capacity is a few orders of magnitude larger than sensor nodes. Back-end devices are typically desktop machines or full-fledged servers that operate at processor speeds of a few GHz, and have a storage and memory capacity of well into the GB range.

All LooCI ports were realised in collaboration with other members of our research team. They are available online [77], together with additional documentation for installation and development. Personal contributions include the implementation of the LooCI/SunSPOT port.

The rest of this section further introduces the prototype implementations of LooCI. For each platform, Sections 4.4.1 and 4.4.2 respectively discuss how the LooCI middleware and LooCI components are implemented. Finally, Section 4.4.3 discusses component development.

4.4.1 LooCI middleware implementations

For each supported platform, an implementation is realised of the LooCI middleware, which includes the configuration engine and event manager. Implemented in each platform's native language, these implementations provide platform-specific, yet, uniform interfaces for component management and event interactions. The configuration engines furthermore implement the event-based management APIs, which results in platform-neutral management via the LooCI management client (see Section 4.3.4). The sole exception to this is component deployment. This requires a specification of the underlying platform of the targeted node, as dynamic code deployment is platform-dependent (see Lines 15 and 17 in Listing 4.3).

Across all implementations, the event bus abstracts away the lower-level details of the networking stack, and is implemented on top of UDP and IPv6. UDP employs a connectionless transmission model that is well suited for loosely-coupled interactions. It furthermore provides a best-effort service that introduces limited overhead, which fits well with the resource constraints within sensor networks. IPv6 in turn facilitates network-level integration between the various tiers that are typically made up out of different IP networks. Within the sensor network tier, IEEE802.15.4 is used at the physical layer and 6LoWPAN provides IPv6 adaptability. While 6LoWPAN and IPv6 provide fragmentation support, LooCI event payloads are typically kept small to fit within a single 127 byte long IEEE802.15.4 frame. Depending on low-level network configuration parameters, such as the 6LoWPAN compression in use, etc., this allows for up to 86 bytes per event.

4.4.2 LooCI component implementations

LooCI does not introduce a new programming language or model for component implementation, but instead reuses those of the respective underlying platforms. As such, LooCI component implementations are platform-dependent. LooCI/Contiki components are implemented in C along Contiki's event-based programming model, while LooCI/SunSPOT and LooCI/OSGi components are implemented in an object-oriented manner using Java. While this limits reuse of a component implementation to a specific platform, it allows to implement LooCI itself as a thin middleware layer. Furthermore, it enables LooCI components to make optimal use of the hardware and software resources provided by the underlying platform. Besides having to adhere to the LooCI communication and configuration APIs, component implementations are essentially free to use whatever functionality is provided by the underlying platforms. This includes software sensor abstractions, digital and analog I/O, timer functionality, data storage, etc.

By consequence also the compilation and dynamic loading of components is platform-specific. However, LooCI puts no additional requirements on the compilation and dynamic loading support of underlying platforms. A prerequisite to successful component deployment across all platforms is however the presence of an operational instance of the LooCI middleware on the targeted node. Compilation of components is performed with standard compilers; AVR-GCC for the Contiki platforms and a standard Java compiler for Sun SPOT and OSGi. Once compiled, a LooCI/Contiki component is encapsulated into a custom binary `.comp` file that adheres to the Executable and Linkable Format (ELF) specification. Run-time dynamic linking and loading of ELF object files is by default supported by Contiki [36]. While this standard ELF support can be used, LooCI/Contiki also offers the possibility to use a more compact, yet proprietary, version. As the supporting AVR platforms are 16-bit platforms, Contiki's default support for the 32-bit ELF variant is rather wasteful. A proprietary 16-bit ELF variant yields a significant 20-30% reduction in component size. LooCI/SunSPOT components are compiled into a `.jar` (Java Archive) file and then further optimised and verified into a deployable binary bundle suited for the Sun SPOT platform. Once deployed, these bundles are executed within Isolates, which allow for the simultaneous execution of multiple applications (i.e. components) on a Sun SPOT node. Finally, LooCI/OSGi components are compiled into `.jar` files that form standard dynamically deployable OSGi-bundles.

```
 1  public class ComponentName extends LooCIComponent {
 2
 3     public ComponentName() {
 4        super(new short[]{ /* list of event types */ },       // provided
 5              new short[]{ /* list of event types */ });      // required
 6        /* optional addition of properties */
 7     }
 8
 9     public void start() {
10        /* logic at component activation */
11     }
12
13     public void stop() {
14        /* logic at component deactivation */
15     }
16
17     /* additional component implementation omitted */
18  }
```

Listing 4.4 – Definition of a LooCI/SunSPOT component

4.4.3 LooCI component development

Although component implementations are platform-specific, the broad lines of component development are largely similar between the various LooCI implementations. Each component definition declares a component type (i.e. name), a set of required and provided interfaces, and a set of properties. Additional application logic is mostly implemented as reactions to event reception, component lifecycle changes, property configuration, or system events like expiring timers, etc. The rest of this section presents more details on component development for LooCI/SunSPOT. Further details on LooCI/Contiki and LooCI/OSGi component development are deferred to Appendix A.

For the Sun SPOT platform, the LooCI component model is mapped to the Java language and component definition thus occurs in an object-oriented manner. As shown in Listing 4.4, LooCI/SunSPOT components are implemented by extending the `LooCIComponent` base class. This base class is part of the LooCI middleware and provides support for component lifecycle management. Additionally, it implements methods for event publication and reception, property management, etc. Component definition is performed by the component's class constructor, shown in lines 3 - 7. A super-call to the `LooCIComponent` class constructor, passes two arrays of event type identifiers.

```java
1 /* Example Property instantiation */
2 Property p = new IntegerProperty(short id, String name, int value);
3
4 /* Implementation provided by the LooCIComponent class */
5 void addProperty(Property property);
6
7 /* Abstract methods declared in LooCIComponent class */
8 boolean updateProperty(Property property);
9 void propertyUpdated(Property property);
```

Listing 4.5 – The LooCI/SunSPOT properties API

```java
/* Implementation provided by the LooCIComponent class */
void publish(short eventType, byte[] data);

/* Abstract method declared in LooCIComponent class */
void receive(short eventType, byte[] payload);
```

Listing 4.6 – The LooCI/SunSPOT event bus API

These respectively enlist the provided and required interfaces of the component. Further implementation of the component is realised by overriding abstract methods provided by the `LooCIComponent` base class; e.g. the `start()` and `stop()` methods in lines 9 - 15 allow for implementing specific behaviour at component (de)activation.

Listing 4.5 shows the properties API for LooCI/SunSPOT. Properties are defined as class variables of data type dependent sub-types of the `Property` class (line 2). They are defined by an identifier and a name, and can contain values of the following types: byte, short, int, long, string, and byte array. Properties are exposed for external configuration via the `addProperty()` method provided by the `LooCIComponent` base class (line 5). This is typically done from within a component's constructor (line 6 in Listing 4.4). Specific logic that needs to be performed before or after updating a property can be implemented by respectively overriding the `updateProperty()` and `propertyUpdated()` methods.

Finally, interaction with the event-bus is implemented by means of the event bus API in Listing 4.6. Publishing an event is done by invocation of the `publish()` method, while event reception is implemented by overriding the `receive()` method. Both are provided by the `LooCIComponent` base class.

A full component implementation is provided in Listing 4.7, which represents the TemperatureSensor component described earlier in Section 4.3.4.

```
package looci.sunspot.components;

import com.sun.spot.*;
import java.io.IOException;
import looci.sunspot.*;

public class TemperatureSensor extends LooCIComponent implements Runnable {

   private ITemperatureInput tempSensor = EDemoBoard.getInstance().
        getADCTemperature();
   private Thread thread;

   private short SAMPLE_FREQUENCY_ID = 1;
   private ByteProperty sampleFrequency = new ByteProperty(
        SAMPLE_FREQUENCY_ID, "sample frequency", (byte)10);

   public TemperatureSensor() {
      super(new short[]{EventTypes.RAW_TEMPERATURE_READING},   // provided
                  null);                                       // required
      addProperty(sampleFrequency);
   }

   public void start() {
      thread = new Thread(this);
      thread.start();
   }

   public void run() {
      Thread thisThread = Thread.currentThread();
      while (thisThread == thread) {
         try {
            int temperature = tempSensor.getValue();
            temperature = (temperature < 512) ? (temperature / 4) :
                 ((temperature - 1024) / 4);
            PayloadBuilder pb = new PayloadBuilder();
            pb.addShort((short)temperature);
            publish(EventTypes.RAW_TEMPERATURE_READING, pb.getPayload());
            Utils.sleep(sampleFrequency.getValue());
         } catch (IOException ex) {
            ex.printStackTrace();
         }
      }
   }

   public void stop() {
      thread = null;
   }
}
```

Listing 4.7 – The TemperatureSensor component implementation in LooCI/SunSPOT.

4.5 Discussion

As stated in Section 4.2, LooCI combines the principles of component-based software engineering and the publish-subscribe communication paradigm. Both technologies are applied in a complementary manner to realise a run-time configurable component model with explicit support for distributed interactions. This section reflects on LooCI within the larger scope of this dissertation, and points out how it contributes to greater flexibility with regards to building applications for sensor systems.

First, LooCI's component model allows for run-time deployment of application components on sensor nodes in the field. Explicitly defined component interfaces and inspection support, enable third-party users to identify prior deployed components and reuse them within new compositions. Furthermore, the behaviour of components can be adapted to the local context or application requirements by means of configurable properties.

Second, in contrast to the state-of-the-art, LooCI comes with support for distributed interactions that is embedded within the component model. By means of the distributed event bus, components can not only interact locally, but just as easily with components on other nodes. This is realised without any centralised event broker, which reduces network traffic. The publish/subscribe-based operation furthermore enables loose-coupling between components, which is of great value in sensor system environments where network connectivity is uncertain, and third-party behaviour unpredictable.

Third, the combination of deployable components and the distributed event bus is very powerful in terms of configurability. LooCI allows external parties to configure individual bindings between specific components. This is a great improvement over state-of-the-art WSN component models that, in the best case, only support local dependency resolution during component deployment. Such fine-grained control brings great freedom and flexibility in specifying which components, interfaces and nodes should interact within a composition.

Fourth, LooCI is defined in a platform- and language-agnostic manner, and has been implemented for several underlying platforms. Additionally, component development in LooCI requires little adaptation by developers that are familiar with those underlying platforms. Consequentially, while component development remains close to the procedural and object-oriented programming paradigms adopted from the respective underlying platforms, the semantics of component implementation are common across all platforms. This greatly facilitates development of cross-platform applications, as described in the evaluation in Chapter 7.

In terms of the service-oriented approach that this dissertation aims to adopt, LooCI provides a light-weight service abstraction in the form of component interfaces. Both provided and required interfaces can be envisaged as the provided and required services of a component. In line with the service-oriented principles, these are furthermore platform- and implementation independent.

To realise fully open sensor systems, LooCI does however have a number of short-comings. First, while inspection support is provided by LooCI, this requires a-priori information about which component or node to inspect. In a large scale deployment, this can be laborious to find out, and more efficient discovery support is required. Furthermore, only component-related information can be inspected for, while additional context related information, like node position and remaining energy, is of equal importance. The contributions in the following chapter provide such apt support.

Second, while component interfaces are typed and only exchange events of equal types, this is only based on simple semantic identification of those interfaces/events. No support is provided to event payload definition, which by consequence is typically realised in an ad-hoc manner, without coordination between various partners. TalkSens, as introduced in the Chapter 6, greatly improves upon this practice.

4.6 Summary

This chapter presented the first contribution of this dissertation; i.e. LooCI, or the Loosely-coupled Component Infrastructure. LooCI features lightweight abstractions for application-level modularity and distributed interactions, which can be inspected and configured at run-time. Its various implementations enables developers to use the same set of abstractions and management APIs to build distributed applications across all tiers of a sensor system.

Section 4.1 provided a detailed list of requirements towards application-level modularity and distributed interactions within open multi-purpose sensor systems. Section 4.2 discussed how component-based software engineering and the publish/subscribe communication paradigm greatly contribute towards meeting those requirements. Section 4.3 presented LooCI itself and discussed its component model, its distributed event bus, and its extensive support for run-time configuration and inspection. Additionally, a short example showed how configuration of a distributed sensor system is performed when using LooCI. Section 4.4 continued with a description of the various implementations of LooCI and presented how components are implemented for LooCI/SunSPOT. Finally, Section 4.5 highlighted some of LooCI's features within the greater scope of this dissertation and pointed out some shortcomings that will be addressed in the following chapters.

Chapter 5

Discovery of application services in open sensor networks

This chapter presents the second contribution of this dissertation; SDLITE, a lightweight status-aware service discovery solution, which adapts the service discovery process to dynamic sensor network systems. To diversify between the multitude of nodes that provide identical services within a sensor network, SDLITE takes their operational and environmental status into account during the service discovery process. In realisation thereof, SDLITE provides a reusable lightweight solution for sharing status information across software components and layers.

Section 5.1 describes the problem that SDLITE solves and highlights shortcomings of existing solutions. Section 5.2 presents the specific requirements on service discovery in open sensor systems. Section 5.3 describes SDLITE itself, and 5.4 presents a proof-of-concept implementation.

5.1 Including node status information in the discovery process

Run-time deployable application logic, as proposed in the previous chapter, and the possible introduction of new nodes within a sensor system, create a need for application-level service discovery. This enables possible future clients to become aware of newly available third-party services, which greatly improves the possibility of reusing those services.

Discovery of services based solely on their functional semantics (e.g. providing a temperature sensing service) is however insufficient in typical sensor networks. After all, due to the nature of sensing applications, and the limited diversity in functionality they exhibit, often a multitude of nodes provide a similar service. To diversify between those nodes, their current operational and environmental status needs to be taken into account. By including parameters such as location, remaining energy, provided sensors, etc., within the service discovery process, more practically useful discovery results can be obtained.

While many status parameters are already monitored by various software modules on a sensor node, they are typically not shared across module boundaries. For instance, while application components read out sensors and determine the node's position, middleware and operating system services maintain overviews of neighbouring nodes, and memory and energy use; yet, each module collects this data for its own purposes, and sharing of data is either not catered for, or provided by module specific APIs. While the first prevents external access to this data, the second results in explicit inter-module dependencies, often across various layers. To facilitate sharing of operational and environmental status information across software modules and layers, central and generic access is needed. This enables not only service discovery, but also a range of other services, such as group management and policy enforcement, to leverage the shared data.

As discussed in Chapter 2, the service-orientation paradigm has been widely applied within sensor networks. Prior service discovery solutions however often do not consider the operational and environmental conditions in which a service operates (µSMS [46], nSOM [47], Servilla [51], WSN-SOA [101]). Others, on the other hand, do take such information into account (Dioptase [14], USEME [22], OASiS [97], TinySOA [152]), but gather it for discovery purposes only. The current state-of-the-art in WSN research, however, acknowledges the benefits of incorporating status information into other system services; examples include application and system management (Chi [49], QARI [69], PMA [115], FiGaRo [127]) and group communication (Logical Neighbourhoods [124], Hood [182]). Yet also here, status information is mostly collected on an individual basis, not shared with other services [49, 69, 115, 124, 127, 182], and only supports a limited or compile-time defined set of parameters [124, 127, 182].

The goal of the research presented in this chapter is to combine the best of the previously mentioned approaches; (i) easy sharing of status information across software modules in a generic manner, and (ii) incorporating this into a lightweight status-aware service discovery solution.

5.2 Service discovery and status sharing requirements

To fulfil the goal stated in the previous section, the proposed solution has to meet the following requirements:

1. **Shared representation of services and status parameters.** A representation needs to be defined that allows developers to specify extensible sets of services and status parameters and present them in a compact format. This must result in a shared namespace that enables all parties to refer to each other's services and status parameters.
2. **Status information sharing across software modules and layers.** A mechanism needs to be provided that allows software modules to share their status data, and to query similar data from others. To limit inter-module dependencies and accomodate run-time software deployment, providers and clients of this data need to be decoupled. Finally, data providers need to retain control of their data.
3. **Status-aware service discovery mechanism.** Based on the solutions to the previous requirements, a service discovery mechanism must be developed that takes into account both service and status specifications. This includes the definition of service discovery message structures and an accompanying query resolution process.

To be applicable within a resource-constrained sensor network environment, these solutions must furthermore take into account the cross-cutting requirement of being lightweight in terms of memory, communication and processing overhead. The following section discusses how SDlite meets these requirements.

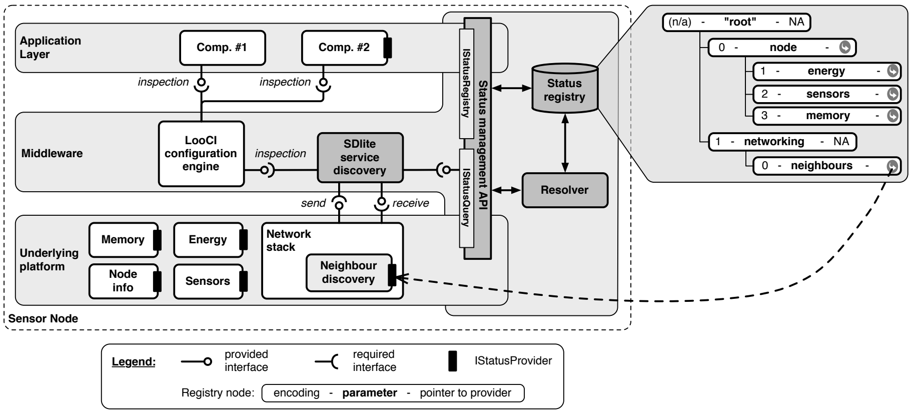


Figure 5.1 – SDLITE interacts with the present application service platform (i.e. LOOCI) and a local status repository to provide status-aware service discovery.

5.3 SDlite: status-aware service discovery

SDLITE provides on-demand status-aware service discovery within open sensor networks. A fully distributed architecture is provided in which each sensor node serves as its own service and status registry. The node-local architecture of SDLITE is shown in Figure 5.1. This distributed solution is favoured over a service registry that is positioned at a central location in the network, and proactively gathers service and status information of all nodes. After all, the variable nature of status data prevents the economic updating of a central registry. Although no messages are pro-actively exchanged to keep the central registry up-to-date, each service discovery action does require dissemination of service requests across the network in search of service providers. As this takes place in an on-demand fashion, the exchange of messages is more valuable than potentially unnecessary pro-active updates of a central registry.

While conceptually SDLITE provides a platform independent solution, it makes some assumptions in terms of service provisioning. Primarily it assumes an application service platform to be at hand that identifies services in one way or another, and provides access to an overview of the available services at a sensor node. To fit within the larger scope of this dissertation, SDLITE is presented here in a version integrated with LOOCI. The interfaces of LOOCI components hereby function as the service abstraction, and services are thus identified by the event types they produce or consume. Additionally, the service discovery module interacts with the LOOCI configuration engine to retrieve information regarding the available services via component inspection.

Status information is shared in SDLITE via the status management API, as shown in the figure. This provides centralised access to the registered status information that is shared by various software modules across all layers of the software stack. Typically, operating system and middleware modules provide operational data like node hardware and software details, energy and memory statistics, and networking information. Environmental data like sensor readings and location data can be provided by system modules or application components, while the latter can also share more application specific data.

The following sections discuss SDLITE in more detail. Section 5.3.1 describes the employed service and status parameter representation. Section 5.3.2 presents the architecture of the status information registry and its API. Section 5.3.3 describes the service discovery process.

5.3.1 Service and status parameter representation

For multiple parties to be able to reason over the same set of services and status parameters, a shared namespace is needed. SDLITE defines the necessary primitives to specify such a namespace and represent it in a compact manner.

An SDLITE namespace is fully defined within the resource-rich back-end and specifies the set of services and the set of status parameters that are used within a system, together with lists of supported data types and operators. Services are identified by a name and have a more compact numerical encoding. As mentioned earlier, LOOCI component interfaces serve as the service abstraction. Consequentially, the SDLITE namespace contains a list of human-readable event type identifiers with their numerical 2-byte encoding.

Status parameters are organised in a hierarchical manner as shown in Figure 5.2. The tree structure allows for (i) intuitive organising of status parameters, (ii) dynamic addition of new status parameters, and (iii) fast retrieval of data by hierarchical search. Internal nodes in the tree represent a set of related status parameters, which are individually represented by its respective leaf nodes; e.g. the `energy` node groups all energy related parameters, such as `energy source`, `vcc`, etc. Internal nodes therefore typically refer to a specific software service, as shown in Figure 5.1, however, they can also be virtual (i.e. not linked to a specific status provider) with the intent of improving the structure of the tree.

Each status parameter within the tree contains a human-readable name, an encoding that represents its hierarchical position, and an associated data type. For example, the `light` status parameter is uniquely identified by

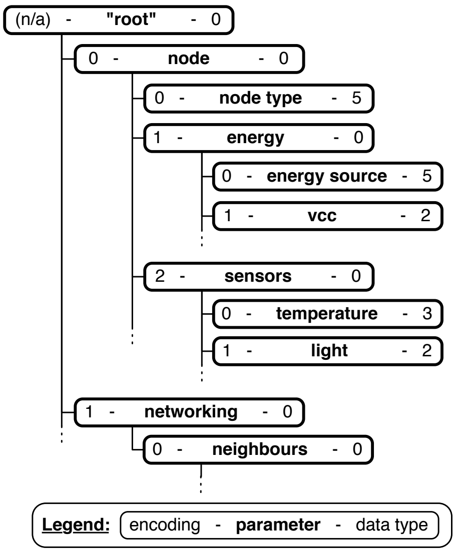


Figure 5.2 – The SDLite status parameter tree is a back-end representation of all status parameters that are known within a sensor system.

`/node/sensors/light`, which in encoded form is `021`. The hierarchical encoding format utilises a single byte per level of the tree. Per parent node, this allows for up to 256 child nodes per parent node. Such numerical encoding reduces overhead in three ways; i.e. communication, memory and processing. Sensor nodes are not user-centric devices (e.g. no visual display) and thus typically rely on a more resource-rich back-end for user interaction. As such, communication, memory and processing overhead can be reduced by replacing human-readable strings with more compact numerical identifiers to exchange and store status parameter information. For user-friendly interaction, back-end tools can resolve the numerical identifiers to their human-readable counterparts. Furthermore, traversing the status tree can be performed more efficiently in terms of processing when it involves single-byte comparisons instead of string comparisons.

The lists of data types, and relational and logical operators supported in SDLite are shown in Table 5.1. The data types are used to uniquely format status parameter values, which allows for their correct interpretation by clients. The operators allow for specification of constraints on status parameters by means of status predicates that are included in service requests; e.g. `/node/node_type` = `spot & /node/memory/free` > `3000`. To do so compactly, both data types and operators are encoded into a numerical representation, which is simply a sequentially assigned integer value.

Data types			**Operators**	
Data type	**Encoding**	**Size (bytes)**	**Operator**	**Encoding**
undefined	0	n/a	undefined	0
byte	1	1	$>$	1
int32	2	4	$<$	2
float	3	4	$=$	3
bool	4	1	$\neq$	4
string	5	n/a	&	5
array	100	n/a	|	6

Table 5.1 – SDLite specifies lists of data types and operators that are used to specify parameter values and status predicates.

On-node awareness of the namespace

Instead of the entire namespace, as it is defined in the back-end, individual sensor nodes are only aware of services and status parameters provided by locally running software. The available services are specified by the deployed application components, while the status parameters are specified by the locally available status providers. As shown in Figure 5.1, services are determined via component inspection, and each sensor node maintains an individual data tree that references the status providers on that node. Each status provider itself, in turn, maintains a list of status parameters it shares, yet only in their compact numerical encoded format. The operational details of status provisioning are discussed later in Section 5.3.2.

Extending the namespace

Both the set of services and the status parameter tree can be extended to include new items. Adding services is straightforward and involves adding the new service to the existing list of services and assigning it an unused numerical identifier. Within LooCI, this means specifying a new event type. Adding a status parameter involves inserting a new node in the status parameter tree. This can be done at all levels but the root level, as long as the number of existing nodes on that level does not exceed the maximum number of 256. Any unused encoding at the respective level can be assigned as the new status parameter's numerical encoding.

To avoid encoding inconsistencies, an append-only policy is applied to the status parameter tree. This means that numerical encodings can only be used once per set of siblings, and that even when their respective node is no longer in use,

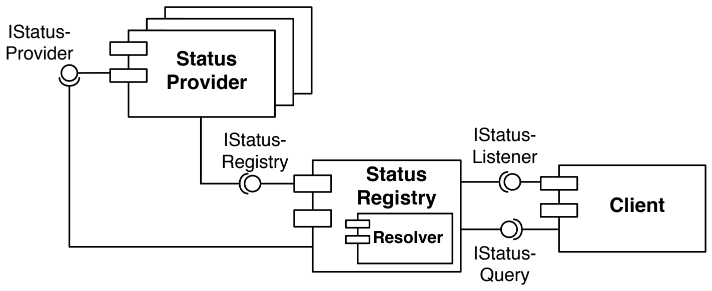


Figure 5.3 – Architecture of the SDLITE status registry.

they cannot be reused. Unneeded nodes are therefore never removed from the tree, rather they are marked as obsolete. This append-only policy is needed as (old) encodings might still be in use by deployed software. Any reuse of an encoding will thus result in that encoding no longer referencing a unique status parameter.

To eliminate obsolete status parameter encodings, a full re-encoding of the tree can be performed. This is however only rarely advisable as it requires all software with references to existing status parameters to be refactored and redeployed. This includes all status providers and clients, which might be over-the-air deployable (e.g. LooCI components), but might also be part of pre-deployed middleware or operating system services. This needs to be avoided as much as possible within operational sensor systems as it is a highly disruptive action that consumes considerable resources within the sensor network.

5.3.2 A central status registry

Sharing status information across software layers, while decoupling status providers and clients, demands a central service in the software stack. Such a service can mediate between status providers and clients and eliminates not only direct dependencies between them, but also their need to implement data sharing mechanisms of their own.

SDLITE provides a central status registry that implements the registry pattern [59]. While this pattern is often applied in object-oriented programming to store objects for later reuse, it is applied here to store cross-layered references to software modules that act as status providers. It therefor maintains a hierarchical data structure as shown in Figure 5.1. Within this tree structure, each node contains a numeric identifier that ties it to the status parameter namespace, and when applicable a reference to a status provider that shares the

```java
public interface IStatusProvider {
   public Object getStatus(byte parameter);
}

public interface IStatusRegistry {
   public void register(byte[] path, byte name, IStatusProvider provider);
   public void unregister(IStatusProvider provider);
}

public interface IStatusQuery {
   public Object getStatusParameter(byte[] path);
   public boolean boolResolve(byte[] predicate);
   public Result resolve(byte[] predicate);
   public void registerStatusListener(byte[] path, long interval,
        IStatusListener listener);
   public void unregisterStatusListener(byte[] path,IStatusListener
        listener);
}

public interface IStatusListener {
   public void statusUpdate(byte parameter, Object value);
}
```

Listing 5.1 – Definition of the IStatusProvider, IStatusRegistry, IStatusQuery and IStatusListener interfaces of SDLite in Java.

```java
...
statusRepository.register(new byte[] {NODE}, ENERGY, new Energy());
...
public class Energy implements IStatusProvider {
    public Object getStatus(byte parameter) {
        switch (parameter) {
            case BATT_LEVEL :
                return Integer.valueOf(Battery.getBatteryLevel());
            ...
        }
    }
}
```

Listing 5.2 – SDLite status provisioning on Sun SPOT.

respective status parameter(s). This results in a very compact representation with minimal memory overhead.

The status registry interfaces support three types of interaction; *registration of status providers*, *status provisioning* and *status querying*. Dedicated interfaces support these interactions between status providers, the status registry and its clients, and are depicted in Figure 5.3 and Listing 5.1. To be incorporated in the status tree, status providers implement the `IStatusProvider` interface and register a reference to it via the `IStatusRegistry` interface of the registry (see Listing 5.2). This makes their status parameters available to the status registry, and hence its clients. The `IStatusQuery` interface allows clients to retrieve that status data. This can either be a one-time operation or periodic updates can be retrieved by means of a registered `IStatusListener` interface. Additionally, more elaborate status predicates can be resolved to be true or not, optionally also returning the actual status parameter values specified in the predicate.

Knowing only the encoding of a status parameter and unaware of its provider, clients can thus use the status registry to retrieve that parameter's value. In turn, the registry looks up the registered pointer to the respective status provider, retrieves the current value of the status parameter therefrom, and passes that value to the requesting client. This separation of interactions through an intermediary status registry decouples status providers and clients. Additionally, status parameter values are only retrieved on-demand and are not pro-actively duplicated. While the status parameters are thus globally retrievable, each status provider remains in control of its own data and direct access to it is restricted to the status registry only.

The status registry as an added platform abstraction

Orthogonal to the posed requirements, the SDLITE status registry can be applied as an additional abstraction over heterogeneous hardware platforms. It hides platform and implementation differences behind its unifying API and namespace, which allows for more generic and reusable applications and services to be developed. For instance, instead of using hardware specific sensor drivers, a LooCI temperature sensing component can retrieve temperature data via the status registry API. This allows the same temperature sensing component to be deployed on various sensor nodes with different hardware sensors and accompanying drivers. Similarly, differences in energy, memory and even network APIs can be abstracted away. This can substantially reduce the development overhead and management complexity in a multi-platform sensor system.

5.3.3 The SDlite service discovery process

The SDLITE service discovery process involves direct interaction between clients and service providers. Clients express their interest in a service by broadcasting a service request message into the sensor network. Upon arrival, each node evaluates the request and, if it can be complied, sends back a service reply message. A number of options enables the clients to influence the discovery process; e.g. in terms of strictness and redundancy. This section shortly describes the service discovery process; the associated service request and reply message formats are described in detail in Appendix B.

An SDLITE service request specifies the service that is being discovered in the form of a numerical service type identifier (i.e. a LOOCI event type) and a status predicate that describes the operational and environmental status that a suitable provider needs to adhere to. It furthermore specifies whether all status parameters in the predicate need to be evaluated, or whether those that are unknown at a resolving sensor node can be ignored. This accommodates for platform differences, and increases the number of nodes that can (partially) fulfil the disseminated request. Finally, the request specifies whether a single suitable provider is sought for or all such providers in the network. This reduces the networking overhead by not forwarding the request when only a single service instance is required and has beeen found. As such, a variety of possible discovery results can be retrieved, ranging from all partially complying services in the network, up to a single fully compliant service.

Upon reception of a service request, the SDLITE process on a node first evaluates whether the requested service is provided by any of the local (LOOCI) application components. This takes place by means of inspection of the locally available components. Only if a service of the specified type is available, an evaluation of the status predicate follows. Status parameter resolution involves retrieval of the current values of the status parameters specified in the status predicate, and evaluation of those values with their counterparts in the predicate according to the associated operators. In case the status predicate is positively evaluated, a service reply message is returned to the originating client. This contains the list of matching components and, optionally, a list of the status parameters specified in the respective request, and their actual current values.

Further details with regards to the service discovery resolution process are discussed in the following section, which presents a proof-of-concept implementation of SDLITE.

5.4 Proof-of-concept implementation

Proof-of-concept implementations of both the service discovery part [170] and the status registry part [169] of SDLITE were realised to demonstrate their feasibility. They were implemented on the Sun SPOT platform [135] and integrated with the respective LOOCI port.

To limit networking overhead, SDLITE service discovery was additionally integrated with the Sun SPOT provided implementation of the Ad hoc On-Demand Distance Vector (AODV) routing protocol [142]. This combines the service and route discovery processes into a single discovery round to leverage the fact that once a service is discovered, the potential client will also need a route towards the service provider. The integration of service and route discovery has previously been proposed in the related literature [28, 53, 91].

The rest of this section first shortly introduces AODV and then presents the SDLITE/AODV integration.

AODV operation

AODV is a MANET (Mobile Ad hoc Network) routing protocol mostly applied in wireless networks with volatile and short-lived routes due to node mobility and intermittent wireless connections. When a route to a certain destination is needed, a route request indicating that destination's address is broadcasted over the network. Any node that either has this address configured as his own or that has an active route to such a node replies to this request. It creates a route reply packet and unicasts it back hop-by-hop to the requesting node. As a result, all intermediate nodes know via which neighbouring node they can reach the required destination, hereby creating a route to it. Hence, the SDLITE service discovery process is similar to the AODV route discovery process with the difference that instead of looking for a route to a node, it looks for a node providing a specified service.

Service and Route Discovery Integration

The SDLITE/AODV integration mainly involves piggy-backing of service discovery messages on their route discovery counterparts and added middleware logic to resolve status-aware service requests. Practically, SDLITE service discovery happens as follows.

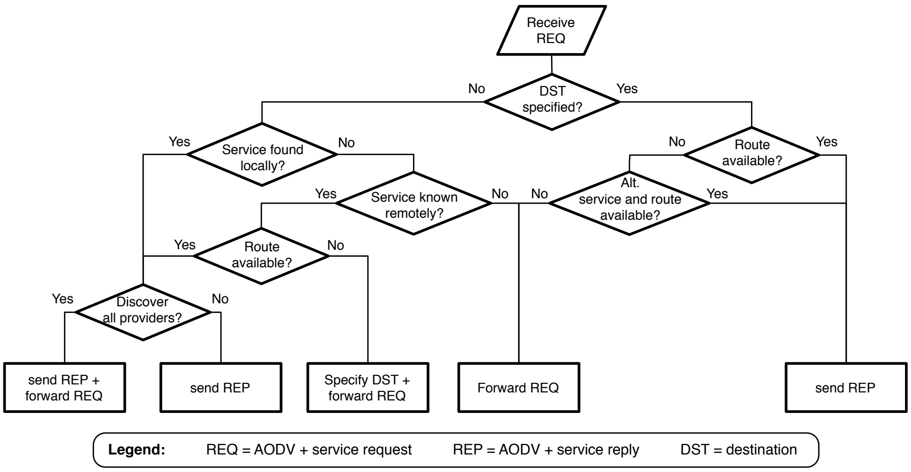


Figure 5.4 – The integrated service and route resolution decision tree.

Clients requiring a service piggyback an SDLITE service request to an AODV route request, which doesn't specify a destination address as at this stage it (i.e. service provider) is still unknown. This request is broadcast into the sensor network, and when received, sensor nodes execute the service and route resolution decision tree as shown in Figure 5.4.

As at this stage no destination address is specified, the left part of the decision tree is evaluated. This part essentially performs *service discovery* to locate complying nodes. It first checks whether a suitable service (including status description) is locally available. If so, the All-flag in the service request is checked, which specifies whether all possible providers in the network are to be discovered. If set, two actions take place. First, a reply is sent hop-by-hop to the originator of the service request, i.e. the service discovery client, with the respective service provider's address specified in the route reply. This reply additionally specifies all local LooCI components that provide the requested service, and, optionally, a list of respective status parameters and their current values. Every intermediate node receiving the reply processes the contained service and route information. The service-related information in the service reply is stored in a service table and the routing information in the AODV routing table. This information is optionally cached during the specified TTL in the reply and is used to resolve future service and routing requests. The second action that takes place when the All-flag is set, rebroadcasts the request so that additional services can be found. If the All-flag is not set, only the reply is sent to the service discovery client. In case a service is not locally found, a locally cached service table is checked for matching entries of remotely provided services. If a remote service is found, and a route is available to it, a reply is send back to the client, and the request forwarded depending on the All-flag. In case a route is not known, the destination address is filled in into the request, and the latter is forwarded in search of a route. In case no matching remote service is known, the request is simply forwarded.

Re-broadcasted requests with a specified destination field, trigger complementary AODV *route discovery*. This involves the evaluation of the right side of the decision tree. If a node has a route available to the specified address, a reply is sent back to the client, hereby completing the end-to-end route. If a route is not available, but the processing node is aware of another compliant service provider, a reply is transmitted pointing at this alternative node. Otherwise, the request is rebroadcasted as it was.

Once a service reply reaches back to the requesting client, its service discovery process takes act of the discovered service. If in the respective request the All-flag was set, the process waits until a timer elapses to allow other replies to arrive before notifying the discovery initiating process of all discovered providers. In case the All-flag wasn't set, the service reply is immediately reported to the initiating process and the associated timer is cancelled. It is possible however that no replies are received in which case the initiating process is notified hereof after the timer elapses.

5.5 Discussion

This section critically reflects on SDlite and its implementation within the larger scope of this dissertation.

First, SDlite complements LooCI's run-time inspection of *known* application components with critical support for the discovery of *unknown* application components. Within an open sensor system in which multiple parties manage various components, network managers must be able to keep an overview of which applications are operating where, how, and under which conditions. The combination of discovery and inspection greatly contributes to obtaining such an overview.

Second, while status sharing was presented in this chapter in support of service discovery, the SDlite status registry provides a generic and centralised API. This allows for status sharing to be applied also within other supporting services such as group management, policy enforcement, resource management, node discovery, etc. Such a shared supporting service reduces resource consumption with respect to otherwise duplicate implementations, and stimulates further integration with other supporting services by means of the open namespace.

However, while the presented discovery solution enables the localisation of services, it does not provide additional information on how to use those services; i.e. how to correctly interpret or structure the respective event contents. In realising a lightweight solution, SDLITE makes use of very basic service identifiers in the form of event type identifiers. While they provide a notion of the event's semantics, they provide no information with regards to their payload contents. The following chapter discusses TALKSENS, which moves away from the opaque event payloads in LOOCI and provides support for the explicit definition of event payloads and run-time retrieval thereof via component inspection.

Finally, while resulting in a lightweight representation, the numerical encoding of status parameters is rather naive. Disadvantages of the proposed encoding include (i) a 1-byte overhead per ancestor in the status tree, (ii) an inherent limit of 256 nodes per parent, (iii) sequential byte-per-byte parsing of the encoding to determine the actual parameter being identified. TALKSENS provides a more compact and clever encoding of such hierarchical information.

5.6 Summary

This chapter presented SDLITE; a status-aware service discovery solution for open sensor networks. SDLITE facilitates discovery of application services based not only on a specification of service functionality, but also takes into account the operational and environmental conditions in which these services operate. In support thereof, a status registry provides consistent sharing of status data across software modules and layers on a sensor node.

Section 5.1 presented the problem of service discovery in sensor networks and highlighted that operational and environmental status information needs to be included to diversify between the many providers of similar functionality. Section 5.2 continued with a detailed list of requirements to such service discovery support. Section 5.3 presented the solution put forward by SDLITE. This included a description of the service and status parameter namespace, the node-local status registry with centralised generic access, and the service discovery process. Section 5.4 presented a proof-of-concept implementation of SDLITE in which service discovery is integrated with the route discovery process to reduce networking overhead. Lastly, Section 5.5 discussed SDLITE's position within the larger scope of this dissertation, and highlighted some additional concerns with regards to service descriptions and the compact representation of hierarchical information.

Chapter 6

Coordinated messaging in sensor systems

This chapter presents the third contribution of this dissertation; the TalkSens message definition framework. TalkSens provides a systematic approach to message definition that results in improved coordination among the various parties involved within a sensor system. Development-time features such as a message description language and message-specific serialisation code generation encourages its use to developers, while subtyping reduces the data flow configuration overhead. Additionally, run-time inspection of components for message descriptions facilitates third-party (re)use of those components.

Within this chapter, Section 6.1 introduces the problems that TalkSens solves and discusses the general principles behind its solutions. Section 6.2 reports on TalkSens's message subtyping support, and Section 6.3 discusses its development-time and run-time support for coordinated message definition.

6.1 Introduction

The correct exchange of meaningful messages is a vital element of distributed applications. Realising this within an open system with multiple parties involved, requires a non-trivial coordination effort to define the various types of messages that are known within the system, and enforce their consistent use.

The contributions presented in this chapter focus on two problems associated with message exchange within open sensor systems. On the one hand, message flows within such systems can quickly become complex as an extensive set of software modules that are deployed on various sensor nodes, exchange a multitude of different message types. Management thereof is challenging, as maintaining a clear overview of message flows is difficult when details about individual interconnections cannot be abstracted away.

On the other hand, the open use of the sensor system infrastructure and re-use of third-party application logic requires a systematic approach to message definition. Ad-hoc message definition, as is largely the state-of-the-art in WSN systems, is no longer feasible and a coordinated approach is required to uniformly define message types and contents. Such support needs to account for the involvement of multiple parties, their varying preferences for data representation, and diverse hardware and software platforms. Furthermore, changing application requirements and software evolution can over time change the contents, and even types, of messages that are exchanged within a sensor system. Dedicated development-time and run-time support for this coordinated message definition is required.

This chapter presents TalkSens, a message definition framework that brings such support to sensor systems.

On the level of message semantics, TalkSens provides support for message subtyping. Message types can be organised into a hierarchy, which allows for reasoning in terms of a group of related message types on a more abstract level. This way, the full details of complex data flows can be abstracted away to more clearly reveal the general principles and goals of the distributed application.

On the level of message contents, TalkSens provides a systematic and coordinated approach to explicitly define message contents. Development tools encourage the use of such a systematic approach, which is extended to the run-time by supporting component inspection for message descriptions. In support of application evolution and heterogeneity in users and platforms, TalkSens allows for well-defined variations of message contents in terms of included data values, data types and units of measurement. This improves reuse of application logic across an open sensor system, yet, without obligating developers to adhere to fixed standardised messages.

The rest of this chapter provides a detailed overview of TalkSens and discusses requirements, solutions and related work. Section 6.2 focusses on subtyping and presents an encoding function that incorporates hierarchical information into compact numerical message type identifiers. Section 6.3 presents the message definition framework and its development and run-time support.

6.2 Subtyping of messages

Within a typical sensor system, a large number of communication relationships can exist that relate to various types of information (e.g. application data, system management). For instance, in the smart office deployment [34] at DistriNet, further discussed in the evaluation in Chapter 7, each single office features 101 relationships between 37 software components that run on 8 nodes. These relationships correspond with 31 different types of messages.

Such complex interaction graphs require considerable configuration effort to establish. Within LooCI, for instance, such relationships are created by binding specific interfaces of components together, which results in an extensive list of binding actions to be executed. Within the smart office deployment, for example, 107 LooCI binding actions are needed per monitored office.

One manner of reducing this effort is by grouping relationships. Grouping can be based on the end-points involved, or on the type of information that is exchanged. An example of the first approach, is LooCI's wildcard support for interfaces, components and nodes, which allows to establish one-to-many, many-to-one, and many-to-many relationships, as discussed in Section 4.3.2.

The subtyping approach presented in this section, aims at grouping relationships based on the type of information that is exchanged; or in other words, based on message types. By arranging message types into a *hierarchy* that specifies *is-a*

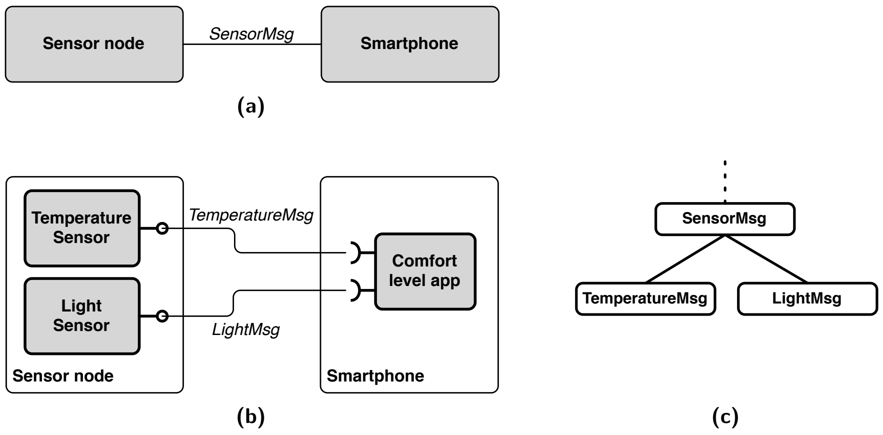


Figure 6.1 – Composition diagrams of a basic smart office application that show various levels of abstraction in data flow *(a,b)*, and the accompanying message type hierarchy *(c)*.

relationships (i.e. subsumption) between its constituent elements, one can reason over groups of elements, called *subtypes*, which are collectively represented by a more abstract element, called a *supertype*. Consider for example the basic smart office application shown in Figure 6.1. Within the message hierarchy shown in Figure 6.1c, `SensorMsg` subsumes all messages that transport basic sensor data; i.e. `TemperatureMsg` and `LightMsg`. This enables reasoning over the application composition on different levels of abstraction. On a higher-level of abstraction, as shown in Figure 6.1a, all sensor data of the sensor node is forwarded to the smartphone, which concisely represents the overall goal of the application. On a more specific level, shown in Figure 6.1b, individual components and bindings appear, which concretely show the existing data flows. This also facilitates application configuration and inspection. For instance, both bindings between the sensor components and the comfort level app, can be configured by a single action; i.e. all messages of abstract type `SensorMsg` must be forwarded to the comfort level app. In conclusion, message subtyping can relieve the developer from dealing with the full details of a complex data flow and more clearly reveal the general principles and goals of the application.

Subtyping has been well applied in message-oriented middlewares. Examples include type-based publish/subscribe systems [43] that rely on the subtyping support of object-oriented languages, and WS-Topics [132] in the Web Services sphere. Both however leverage on technologies that introduce too much overhead to be comfortably applied within constrained sensor nodes; i.e. object-orientation and the Web Services stack. The current state-of-the-art of message typing in constrained environments often provides no support for sub-typing. Examples include TinyOS' default messaging system (Active Messages [21]), publish/subscribe-based systems for WSNs (Mires [161] and TinyCOPS [66]), as well as LooCI [71]. Where subtyping is supported within sensor networks, this is realised in sub-optimal manner by means of verbose textual identifiers or simple numerical encoding techniques. Within CoAP [156], for instance, resources are ordered in a hierarchy and are addressed using human-readable resource-paths; e.g. `/sensor/temperature`, which is slightly improved by the IPSO application framework [155], which proposes shortened path templates (e.g. `/sen/temp`). The OMA Lightweight Machine to Machine specification [134] further compacts CoAP resource paths into a numerical representation (e.g. `/4/0/2`). Both the IPSO and OMA approaches however limit human-readability, while not offering great compaction. NanoSD [93] on the other hand, provides a more compact encoding and makes use of bitstreams, like `01001`, which is parsed into `0.1.0.01` and maps to a `Sensors.Ambient.Weather.Temperature` service identifier. Each sensor node is however required to know the full details of the service tree structure (e.g. amount of children per parent) to be able to correctly parse the bitstream. In conclusion, compact encoding of hierachical information in support of subtyping is missing within the WSN state-of-the-art research.

The rest of this section presents the subtyping support of TalkSens. Section 6.2.1 lists the requirements for subtyping within the resource constraints of sensor nodes. Section 6.2.2 introduces an adapted encoding function for including hierarchical information within numeric identifiers. Finally, Section 6.2.3 discusses how these numeric identifiers are applied within LooCI to enable event subtyping.

6.2.1 Subtyping requirements and approach

The goal of the research presented in this section is to provide nominal subtyping of messages within resource constrained sensor networks. In general, messages have a type, and type-related contents. A message type is defined as the external meaning of a message (e.g. temperature message); it gives meaning to the otherwise arbitrary set of bytes that make up the message's payload. Within nominal subtyping, an explicitly declared link exists between types and their subtypes. This in contrast to structural subtyping, in which the structure, i.e. message payload contents, (implicitely) defines whether one type is a subtype of another. The nominal subtyping discussed in the rest of Section 6.2 only relates to message types. The contents of message payloads, and more precisely the definition thereof, is the topic of Section 6.3.

To limit the communication and storage overhead, message types are commonly identified within sensor networks by means of a numeric representation, e.g. Active Messages [21] applies a one-byte handler identifier to dispatch messages of a certain type to a fitting message handler. To realise subtyping support within these constraint environments, hierarchical information thus needs to be encoded within those numeric identifiers. Such encoding should ultimately enable testing whether one type is a subtype of another within the respective hierarchy.

A hierarchy encoding function must meet the following requirements to suit application within constrained sensor networks:

- **Compact representation.** A compact representation means that an identifier must contain all necessary sub-typing information in an encoding that uses a minimum amount of bytes.
- **Efficient subsumption testing.** Subsumption testing should require (i) minimal computation and (ii) minimal input of hierarchy information.
- **Conflict-free incremental encoding.** Incremental encoding allows the addition of new entries into the hierarchy at any time without requiring

the recomputation of existing identifiers and consequential updates of software that uses the old identifiers.

To meet these requirements, we build-upon a prime number-based encoding function as developed by Preuveneers et al. [146, 147]. The presented adaptation optimises the original function for constrained sensor networks by imposing a single-inheritance restriction upon hierarchies. Previous experiences in building sensor network applications [72, 168], have shown such single-inheritance hierarchies to be sufficiently expressive. Through further clever adjustment of the original encoding function, this results in a drastic increase in compaction of the encoded identifiers.

6.2.2 Prime-based hierarchy encoding function

The presented encoding function is based upon the *fundamental theorem of arithmetic*, also called the *unique prime factorisation theorem* [64] which states the following:

> *Every natural number, different from one, is either prime or, if it is composite, can be represented by a product of prime numbers, which is unique up to the order of factors.*

Less formally, this means that every positive integer greater than one has a unique prime factorization. From this it can be derived that a product of prime numbers is *only* divisible by those prime numbers. It is this property of prime number theory that is applied within the presented encoding function, both in its original and adapted form.

Principles of prime-based encoding

The principle workings of the prime-based encoding function is visualised in Figure 6.2. While the original function supported encoding of multi-inheritance hierarchies, the figure presents a single-inheritance hierarchy as it suffices to explain the encoding function within the scope of this dissertation. Prime-based encoding consists out of three steps; (i) creating a hierarchy of message types, (ii) prime number assignment, (iii) identifier calculation.

Within the first step, message types are organised in a hierarchical tree structure. Each node in the tree represents a single message type. Leaf nodes represent specific message types, while internal nodes represent abstract message types.

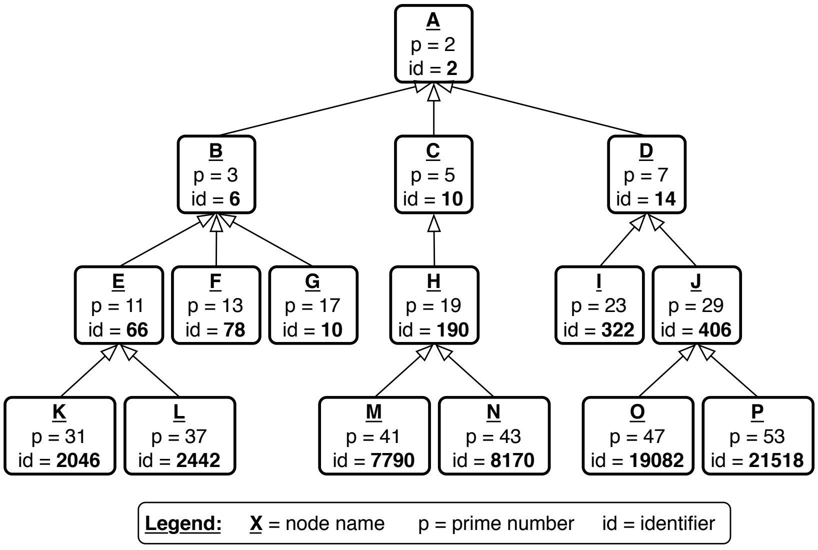


Figure 6.2 – An example hierarchy encoded with the original encoding function.

In the second step, prime numbers are assigned to all nodes in the tree. In the original encoding function, each node is assigned a unique prime number. The sequence of assigning primes influences the overall compactness, or size, of the resulting identifiers. Within the example, a breadth-first prime assignment is applied, which overall results in the smallest identifiers.

Finally, within the third step, the values of the message type identifiers are calculated. For each node, this is equal to the product of its own prime with the primes of all of its ancestors; or consequentially, the product of its own prime with the identifier of its parent. More formal, with T being a node in the tree, p_T its assigned prime number, and id_T its identifier:

$$id_T = \begin{cases} 2 & \text{if T is root} \\ p_T * id_{T'} & \text{where T' is the direct supertype of T} \end{cases} \tag{6.1}$$

Subsumption test

The subsumption test determines whether one node is an ancestor of another. Whether or not this is the case is encoded within the values of their identifiers. Following from the properties of prime numbers, the encoding function causes identifiers to be divisible (i.e. without remainder) only by the identifiers of their ancestors, and no other identifiers. To test whether node A subsumes node B ($B <: A$), the modulo operation can thus be used as follows:

$$B <: A \Leftrightarrow id_B \bmod id_A = 0$$

A proof of the correctness of the subsumption test is given by Preuveneers et al. [146].

Adapted prime number assignment and encoding

Closer inspection of the resulting identifiers in Figure 6.2, reveals that they form a very sparse set of integers. One obvious example thereof is that all odd numbers are omitted from the set. The reason of this sparsity is two-fold; (i) identifier values are the result of a multiplication operation, and (ii) the factors of these multiplications are primes, which themselves form a sparse integer set. Consequentially, the values of the identifiers rise fast with an increasing number of nodes, which critically results in a larger number of bytes needed to represent these values. This contrasts with the compactness requirement stated before.

To reduce the growth in size, a number of optimisations can be introduced to the prime number assignment strategy, under the premise that the hierarchy is restricted to a single-inheritance structure. First, as this results in a tree with a single root, the root can be assigned *1* as its 'prime' value. Although not a prime number by definition, this has no influence on the correctness of the encoding function within a single-inheritance structure. Second, single-inheritance allows for reusing prime numbers in the various sub-trees of the hierarchy. This means that more often identifiers are a product of the lower-value prime numbers and consequentially have a lower value themselves.

To still guarantee correct subsumption testing, the reuse of prime numbers needs to adhere to the following rule. The prime number to be assigned to a node must be different from:

1. the prime numbers assigned to the node's ancestors and their direct siblings,
2. the prime numbers assigned to the node's siblings,
3. the prime numbers assigned to the offspring of the node's siblings, and
4. the prime numbers assigned to the node's offspring.

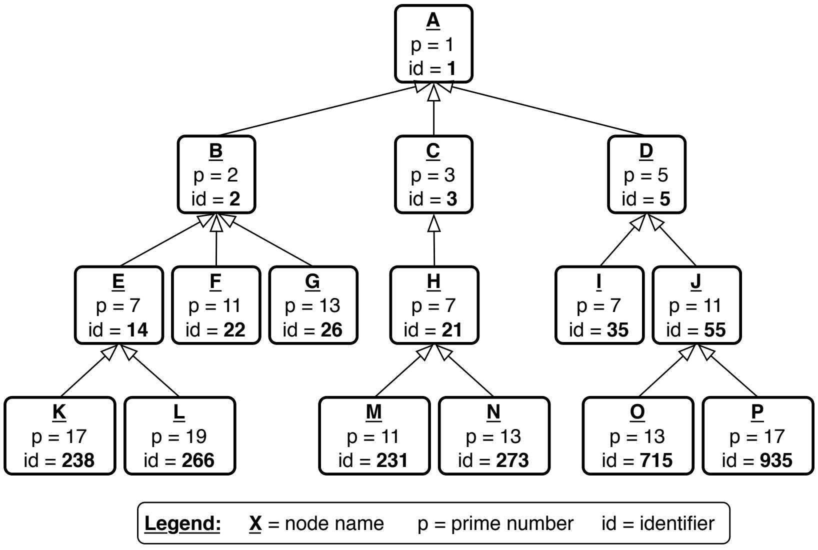


Figure 6.3 – The example hierarchy encoded with the adapted encoding function.

The last requirement is typically inherently met when prime number assignment occurs in a top-down, breadth-first manner. Figure 6.3, provides a concrete example of the prime number reuse and the increased compactness of the resulting identifiers. Notice for instance how *7* is reused as prime number for nodes *E*, *H* and *I*.

For the adapted encoding function, Equation (6.1) can thus be adapted correspondingly to the following. With T being a node in the tree, p_T its assigned prime number, and id_T its identifier:

$$id_T = \begin{cases} 1 & \text{if T is root} \\ p_T * id_{T'} & \text{where T' is the direct supertype of T} \end{cases} \tag{6.2}$$

To prove that the subsumption relationships are preserved by the reuse of prime numbers, this can be restated as follows. Assume a set of nodes in a tree $\chi = \{C_1, C_2, ..., C_n\}$. We define $\Gamma(C_i)$ as the union of C_i's assigned prime and the set of primes of its ancestors. The encoding function that determines a node's identifier can then be written as:

$$id_{C_i} = \prod_j p_j \text{ with } p_j \in \Gamma(C_i) \tag{6.3}$$

By definition of subsumption; C_1 can only subsume C_2 if there is a subtree rooted at C_1 which contains C_2 (possibly as root). This means that:

$$\Gamma(C_1) \subseteq \Gamma(C_2) \tag{6.4}$$

By definition of the encoding function (6.3), each node's identifier is the product of its own prime and the set of primes it inherits. Under single inheritance and reuse of primes, the subsumption test thus remains correct as it will only succeed in case the set of primes of vertex C_2 is a superset of the set of primes of the more abstract vertex C_1, as reversely stated in (6.4). By the rules of our prime number assignment strategy, this can only be the case when C_1 is an ancestor of C_2.

For completeness, the multi-inheritance example in Figure 6.4 shows that the subsumption test is no longer valid when reusing prime numbers. In that case the set of primes of a multiple inheriting vertex can also be a superset of a vertex that only inherits from some of its ancestors. For instance, $\Gamma(E) = \{1, 2, 3, 7\}$ is a superset of $\Gamma(H) = \{1, 3, 7\}$, yet, no subsumption relation exists between E and H in the given hierarchy.

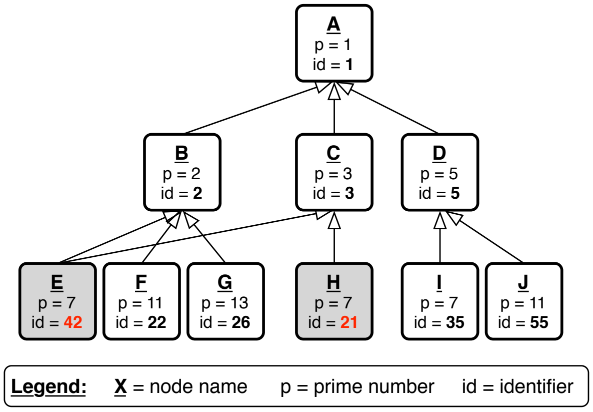


Figure 6.4 – Reuse of primes under multi-inheritance breaks the correctness of the subsumption test, as shown by the identifiers of nodes E and H.

Incremental encoding of late additions to the hierarchy

New message types can be added to an existing hierarchy at all levels, but the root level. By applying the prime number assignment rules stated previously in Section 6.2.2, such addition can be performed without changing the identifiers of the existing message types. An example is shown in Figure 6.5 in which nodes X and Y are added to the hierarchy shown in Figure 6.3.

In contrast, existing nodes should not be removed from the hierarchy. The primary reason for this is that it releases their assigned prime. Subsequent reuse of those primes for the later addition of a new node could lead to errors. False positives during subsumption testing can occur when running applications still refer to the message types of the removed node. In case a message type should no longer be used, the proposed intermediary solution is to retain its node, and possible offspring, as placeholders of the previously used primes and identifiers.

To eliminate deprecated message types and improve compaction of the remaining message type identifiers, a hierarchy can always be fully re-encoded. This includes new prime number assignment and identifier calculation. While this might be beneficial, it should be prevented within operational settings as it changes the identifiers of existing message types within the hierarchy. Consequentially, all software that refers to the *old* identifiers needs to be updated to refer to the *new* identifiers.

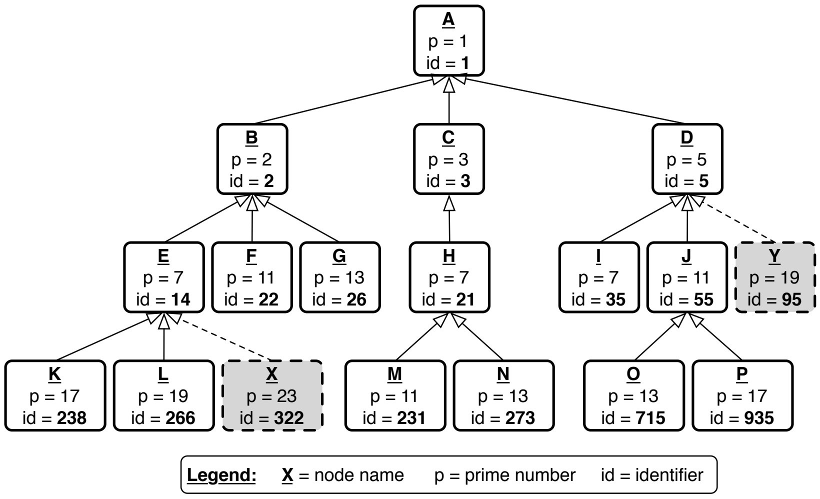


Figure 6.5 – Late addition of nodes X and Y to the example hierarchy.

Impact of the adaptation

The increased compaction of the adapted encoding function results in reduced communication and storage overhead, since shorter message type identifiers need to be included in messages and stored in memory. Importantly however, the extent of the achieved compaction additionally results in reduced computation overhead.

As discussed in the evaluation in Chapter 7, the presented adaptations allow for identifiers in sufficiently large hierarchies to be encoded within less than 8 bytes. This is in contrast with the original encoding function, which quickly results in identifiers of larger size. Consequentially, standard-size integers (i.e. up to 8 bytes) and a standard modulo operation can be used on most platforms to respectively represent message type identifiers and perform subsumption testing. The original encoding function, on the other hand, requires custom integer representations to be implemented and features an extensive subsumption test, involving both the primes *and* identifiers; it quickly tries to rule out subsumption by means of a number of theorems before applying a more computation intensive custom modulo operation [146]. Because of the presented adaptation, no such custom support has to be implemented and subsumption testing can be performed efficiently using only the identifiers. Within the resource constraints of sensor nodes, this is of great importance.

6.2.3 Event subtyping in LooCI

This section reports on the practical application of subtyping within LooCI based on the presented encoding function. Within the following, first the required implementation changes to LooCI are discussed, followed by a number of guidelines on how to practically use subtyping during component development and application management.

Implementation changes

Minimal changes to the LooCI middleware are needed to add subtyping to its binding model. These changes are needed to allow for larger event type identifiers and to enable subsumption testing during event dispatching.

The use of larger event type identifiers affects (i) the serialised event format, (ii) the subscription tables, and (iii) the component interfaces. As they can grow reasonably large, and to avoid unnecessary transmission of leading zero-bytes, a variable size integer encoding is used within the serialised events that are

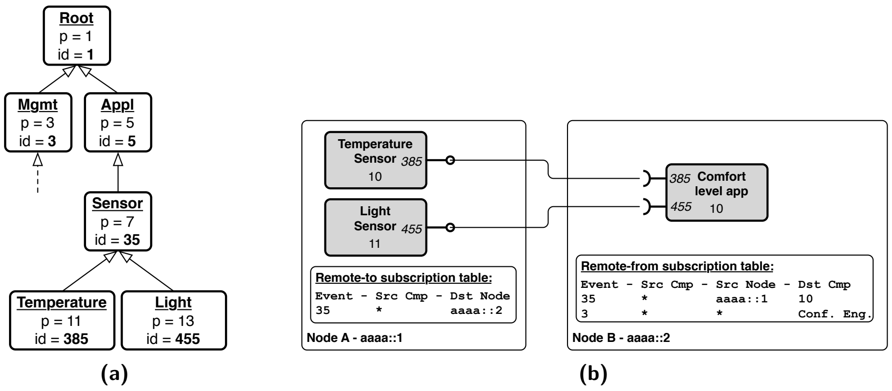


Figure 6.6 – Within LooCI, organising events in a hierarchy *(a)* allows for establishing more intuitive bindings on a higher level of abstraction with fewer entries in the respective subscription tables *(b)*.

exchanged between components. Within this encoding, a first byte specifies the number of bytes that follow to represent the actual value of the identifier. This allows for identifiers to be represented with a maximum length of 255 bytes, which is more than sufficient. Changes to the subscription tables and component interfaces, merely involve refactoring to use adequate data types to represent the larger identifier values. In the current implementations, this is realised by applying the largest size integer data type offered by the underlying platforms. This is typically an 8-byte integer format, which in practice is sufficient, as shown in the evaluation in Chapter 7.

Additionally, LooCI's event dispatching process needs to be adapted to not only dispatch events based on equality of event types but also based on subtyping relations. As the subsumption test merely involves a modulo operation, this requires minimal changes to the existing implementations.

Subtyping in practice: a discussion

Practical use of event subtyping in LooCI is shown in Figure 6.6. Based on the (partially shown) event hierarchy, the earlier presented smart office application is created. As a result of the added subtyping support, the connections between both sensor components and the comfort level app, can be created with a single binding, which states that all `Sensor` events from any (***) components on node *A* need to be forwarded to the comfort level app component (*10*) on node *B*. Additionally, by placing all configuration and inspection events underneath the management supertype (i.e. `Mgmt`), these can dispatched to the configuration engine with a single entry in the remote-from subscription table, as shown for node *B*. This contrasts greatly with the otherwise 31 individual entries needed to dispatch all incoming LooCI management events. Finally, as can be seen in the figure, only the locally needed identifiers are used within the components and subscription tables. The full event hierarchy, which includes additional subtyping information and prime numbers, is not needed on the constrained sensor nodes.

As discussed in Section 6.2.2, fully re-encoding an event hierarchy necessitates an update of all software that refers that hierarchy's identifiers. Figure 6.6 indicates the impact thereof within LooCI, which involves (i) refactoring and redeploying the LooCI middleware, as the configuration engine refers to LooCI management events; (ii) refactoring and redeploying all application components, as their interfaces refer to event type identifiers; and (iii) reconfiguration of all bindings with the new event type identifiers. Even for small sensor systems can this be a disruptive and expensive operation.

Finally, while event subtyping within LooCI provides configuration benefits, it can also introduce some complexities and errors. One major reason for this is that only nominal subtyping is provided, with no additional support for inheritance of event contents. More practically, although `Temperature` and `Light` are both of type `Sensor`, the presented subtyping support does not specify that they share a (partial) common payload structure; both event types may contain different sets of values, in different sequences, and expressed in different data types. A solution to some of these shortcomings is presented in the subsequent section, however, additional guidelines are presented here with regards to the use of subtyping within LooCI. These are the following:

- **Serialised events** should always be of a specific (i.e. leaf) event type. It is only at that level that payload contents is specified and can thus be correctly parsed by subscribed components. Furthermore, as abstract (non-leaf) event types have no specific meaning, it makes little sense to instantiate events of those types.

- **Component interfaces** (both provided and required) should be of a specific (i.e. leaf) event type for the same reasons. One exception exists however for components that do not require to parse the event payload. An example is a logging component that simply stores the serialised payload of incoming events. In these cases, the component can include an encompassing generic required interface, which eliminates the need to specify within the component's code all specific event types the component is able to process.

- **Bindings**, or subscriptions, can be made using both abstract and specific event types, which can substantially reduce configuration overhead.

6.2.4 Conclusion

This section presented the first contribution of the TalkSens framework; a hierarchy encoding function that results in compact node identifiers and an associated lightweight subsumption test. This has been applied within LooCI to provide event subtyping on resource constrained sensor nodes. The changes needed therefor were minimal, yet, result in a considerable reduction in management overhead due to the grouping of event types within single binding configuration commands. This will be practically shown in the evaluation in Chapter 7. The following section reports on TalkSens' support for the systematic and coordinated definition of message contents.

6.3 Message content definition

In addition to defining the semantics of messages by means of identifiers, TalkSens provides support for defining message payloads; i.e. the contained values and their format. A strict definition thereof is of great importance within open sensor systems to allow for the correct and meaningful exchange of data between components of various stakeholders. This however requires a systematic approach that enforces developers to describe the contents of the messages and facilitates the consistent use thereof among the various parties involved. Such coordination of message definitions has received surprisingly little attention by the WSN research community. Within the state-of-the-art of sensor network programming, the following set of related problems can be identified.

First, message semantics and contents are often defined in an implicit and ad-hoc manner within source code. This hampers interaction between independently developed application components. Within Contiki, for instance, message payloads are typically created by means of byte array filling, as shown in Listing 6.1. Besides possible documentation, message contents is thereby implicitly defined through implementation. Furthermore, (de)serialisation is done manually, which is particularly tedious and error-prone, and leaves substantial scope for logic errors due to incorrect (de)serialisation. Better serialisation support is provided by Active Messages [21] within TinyOS, as shown in Listing 6.2. Here, custom C-structures are applied to specify message contents and automate serialisation. This is however a platform-specific solution, which cannot be reused across an entire multi-platform sensor system. A

```
uint16_t temperature = read_temperature();
uint16_t timestamp = clock_time();

// Serialisation (and implicit definition) of the message structure
uint8_t buffer[4];
buffer[0] = (uint8_t)(temperature >> 8);
buffer[1] = (uint8_t)temperature;
buffer[2] = (uint8_t)(timestamp >> 8);
buffer[3] = (uint8_t)timestamp;

// Sending a message
simple_udp_sendto(&connection, buffer, sizeof(buffer), &address);
```

Listing 6.1 – Pseudo-code example of the typical byte-array filling approach to message serialisation in embedded systems development, e.g as used in Contiki.

```
// Defining a platform-specific message structure
typedef nx_struct TemperatureMsg {
  nx_uint16_t temperature;
  nx_uint16_t timestamp;
} TemperatureMsg;

// Serialising and sending a message
...
TemperatureMsg* tmppkt = (TemperatureMsg*)(call Packet.getPayload(&pkt,
     sizeof (TemperatureMsg)));
tmppkt->temperature = temperature_value;
tmppkt->timestamp = time_value;
AMSend.send(AM_BROADCAST_ADDR, &pkt, sizeof(TemperatureMsg));
...
```

Listing 6.2 – Pseudo-code example of the C-struct definition approach to message serialisation in embedded systems development, e.g as shown here in the Active Messages approach within TinyOS.

final issue in this regard is that developers often have limited or no access to third-party source code, or documentation thereof. While this prevents easy third-party interaction on its own, it also leaves great space for developers to (re)define messages with conflicting identifiers and contents.

Second, in the face of multiple stakeholders, various incompatible content variations of semantically equivalent messages are bound to exist. Such variations can be the result of cultural preferences, e.g. °C vs. °F; practical reasons, e.g. no floating-point support on constrained embedded devices; and changing application requirements, e.g. adding timing information to previously non-timed sensor readings. Where meaningful, well-defined variation in message contents should be supported to improve (re)use of application components.

Third, within open sensor systems, interactions are often established with software that is only discovered at run-time. The ability to retrieve message definitions directly from the item of interest, i.e. the running software component, would greatly facilitate such interactions as it removes the requirement to study third-party source code or documentation.

To address these problems, TalkSens provides a message definition framework. This comes with development support to define message contents in a platform-neutral manner, and with accompanying generation of message serialisation code. Central administration of message definitions furthermore contributes to the necessary coordination among various stakeholders in an open sensor system. And lastly, run-time retrievable message definitions facilitate interaction with newly discovered software. For the latter, TalkSens is integrated with LooCI, which results in runtime inspection of component interfaces for respective message (i.e. event) descriptions.

The rest of this section presents the TalkSens message definition framework in greater detail. Section 6.3.1 lists the requirements to a message definition framework for sensor systems and provides an overview of the distributed TalkSens framework. Section 6.3.2 presents the TalkSens data model, which underpins message definition support as explained in Section 6.3.3. Section 6.3.4 discusses the TalkSens serialisation format, and Section 6.3.5 presents additional development support. Section 6.3.6 reports on the integration of TalkSens and LooCI, and finally, Section 6.3.7 discusses related work.

6.3.1 Message definition requirements and approach

The overall goal is to provide a framework that facilitates interactions between independently developed application components within sensor systems. Specific requirements to such a framework are the following.

- **Message definition support.** A mechanism is required for the specification of message types and their payloads in terms of values, data types, and where applicable, units of measurement. This mechanism should support well-defined adaptations to deal with changing application requirements; e.g. adding new message types, changing the content of a message, and adding newly supported units of measurement.

- **Development support.** Systematic use of (pre-)defined messages needs to be encouraged at development-time to facilitate the development of correctly interoperable application components. Developers need to be able to conveniently consult the set of defined messages, apply changes when necessary, and be assisted with message serialisation.

- **Run-time inspection support.** Message definitions need to be retrievable directly from run-time components to facilitate third-party interactions.

- **Lightweight sensor network solution.** The proposed solution should consume a minimal amount of sensor network resources, and leverage the available resources at the back-end.

The overall approach of TalkSens to satisfying these requirements is to provide a distributed framework for message definition, as depicted in Figure 6.7. In the resource-rich back-end environment, a *data model* is maintained that represents the currently supported message types, message contents, data types, units of measurement, and language mappings. This data model underpins a *message description language*, that is systematically used at development-time to define message contents. For this, TalkSens draws inspiration from interface description languages (IDL) like Protocol Buffers [57] and Thrift [6]. To suit the WSN-specific needs, support is added for units of measurements and run-time available lightweight message definitions. Additional development support is provided by means of an *IDE plugin* that provides access to a message definition repository and features generation of message-specific *serialisation code*. As such, TalkSens provides a define-once/use-multiple-times mode of operation. The resulting code is to be included in the source code of software that produces/consumes these messages, which simplifies error-free message (de)serialisation. Next to the serialisation code, also a lightweight version of the respective message definitions is embedded within components. These are

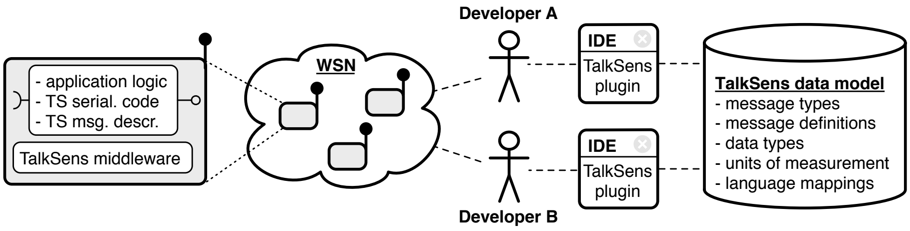


Figure 6.7 – Conceptual diagram showing the distributed architecture of the TalkSens framework.

retrievable at run-time by using the inspection features of run-time configurable *component models* like LooCI. In support thereof, a small *middleware* service is provided on TalkSens enabled nodes. As such, the TalkSens solution maximally leverages the distributed environment; full message definitions are specified and available only in the back-end, while in the sensor network the available information is limited to the minimum that is required by the local software for serialisation and runtime inspection.

6.3.2 The TalkSens data model

The core of TalkSens is formed by its data model that specifies a generic message structure, the data types and units of measurement used to format the data inside message payloads, language mappings and ultimately a set of message definitions. This section discusses the data model, with a description of the generic message structure being deferred to the following section.

The data model provides a number of primitives that are used for message definition; data types with accompanying language mappings, and units of measurement. These are arranged in extensible sets, as partially shown in Table 6.1. The data type list specifies the language-independent primitive data types that can be used within TalkSens to represent values in a message. Except for strings, these are fixed-size data types, which are associated with language-specific primitive data types for each of the supported underlying languages (currently C and Java). Similarly, the units list specifies the units of measurement that can be used to express values that represent a physical quantity. Elements in both lists are mapped to a numeric encoding for compact serialisation.

Table 6.1 – A partial overview of the extensible data type and unit of measurement lists in the TalkSens data model. (*n/a* means 'not applicable')

Data types					**Units**	
Data type	**Enc.**	**Size**	**C type**	**Java type**	**Unit**	**Enc.**
undefined	0	*n/a*	*n/a*	*n/a*	none	0
int8	1	1	int8_t	byte	°C	1
int16	2	2	int16_t	short	°F	2
int32	3	4	int32_t	int	cd	10
int64	4	8	int64_t	long	bar	10
string	5	*n/a*	char[]	String	Pa	11
float	6	4	float	float	ppm	30

Besides these primitives, the data model contains a set of currently defined message types and associated payload descriptions. A message type defines the semantic meaning of a message (e.g. a temperature message), and is mapped to a numeric identifier for compactness. Within the integrated solution of this dissertation, this set of message types takes the form of a hierarchy of LooCI events, as presented in Section 6.2. Furthermore, for each message type, a content definition is specified that defines which data values are part of the message's payload, and which data types and units of measurement are used to represent them.

The TalkSens data model is designed to support extension. Message types, data types and units of measurement can be added to the respective lists and are numerically encoded. An append-only strategy is enforced to prevent hard to detect errors caused by reuse of previously removed encodings. Additionally, language mappings need to be added to fully integrate new data types into the data model, or even to add support for new underlying platforms or languages. Finally, changes to the data model can require a corresponding update of the serialisation code generator (see Section 6.3.5).

6.3.3 Message definitions

Within TalkSens, message contents is defined by means of a message description language (MDL). The TalkSens MDL is a minimal declarative language that allows for the platform-neutral definition of such message contents. Its grammar, as presented in Listing 6.3, enables the intuitive specification of correctly structured message definitions. Examples are shown in Listings 6.4 - 6.7. The inclusion of data types and units of measurement in the syntax forces developers to only use those that are specified within the data model. By means of the language mappings, this provides run-time type correctness during message exchange. Correct use of units is however not enforced, as by default, this is not supported by the underlying languages (i.e. C, Java).

Message definitions contain (i) a human-readable *name*; (ii) a *message type* that semantically defines the message; and (iii) a *data record* that specifies the message contents. The message type relates to those defined in the data model. Data records are composed of a set of *data elements* and/or nested data records. Data elements, in turn, are atomic and represent a data value with a certain data type and unit of measurement. Both data records and elements are given a human-readable name and sequence number. The former serves to generate intuitive serialisation APIs, the latter is used under the hood for element/record identification. Finally, both data elements and records can contain single values or an array of values, as is indicated by the `repeated` keyword (see Listing 6.3).

```
message      = "message" name "{" messageType dataRecord "}" ;
messageType  = "messageType" name ;
element      = dataRecord | dataElement ;
dataRecord   = "dataRecord" name seqno ["repeated"] "{" {element} "}" ;
dataElement  = "dataElement" name seqno ["repeated"] "{" "dataType"
     dataType "unit" unit "}" ;
dataType     = "undefined" | "int8" | "int16" | "int32" | "int64" | "float"
      | "string" ;
unit         = "undefined" | "celsius" | "fahrenheit" | ... ;
```

Listing 6.3 – Extended Backus-Naur Form (EBNF) representation of the TalkSens MDL grammar.

```
message abstr_temperature_msg {
  messageType temperature
  dataRecord temperature_rd 0 {
    dataElement temperature_el 0 {
      dataType undefined
      unit undefined
    }
  }
}
```

Listing 6.4 – An abstract message definition specifies the message type and contents.

```
message simple_temperature_msg {
  messageType temperature
  dataRecord temperature_rd 0 {
    dataElement temperature_el 0 {
      dataType int16
      unit celsius
    }
  }
}
```

Listing 6.5 – A message variant definition completes the definition with specific data types and units.

```
message another_temperature_msg {
  messageType temperature
  dataRecord temperature_rd 0 {
    dataElement temperature_el 0 {
      dataType float
      unit fahrenheit
    }
  }
}
```

Listing 6.6 – Message variants can specify different data types and units.

```
message timed_temperature_msg {
  messageType temperature
  dataRecord temperature_rd 0 {
    dataElement temperature_el 0 {
      dataType float
      unit fahrenheit
    }
    dataElement timestamp 1 {
      dataType int64
      unit ms
    }
  }
}
```

Listing 6.7 – New content can be added to abstract message definitions, and further defined in a variant definition.

Variation within message contents

To account for cultural differences, changing application requirements, and practical limitations, TalkSens supports well-delimited variations between messages of the same type. Therefore, messages can be defined on two levels of detail.

Abstract message definitions describe basic message content, i.e. the required data records and data elements that correspond with the message's semantics. Data types and units are typically not specified, as shown in Listing 6.4.

Message variant definitions are fully specified instances of a corresponding abstract message definition. They are typically used to describe the specific message contents an application component produces or consumes. Multiple variants may exist, as shown in Listings 6.5 and 6.7.

On addition of a new message variant definition, the latter is checked against its respective abstract counterpart for compliance. These different levels of detail allow for messages to vary in terms of *content* and *representation*.

Variation in contents, i.e. the set of data elements and records of a message, is allowed in support of changing application requirements; e.g. new regulations might cause a timestamp to be added to a temperature message, as shown in Listing 6.7. Such variation can only be introduced at the level of abstract message definitions and only allows for addition of new data elements and records, not removal. This ensures consistency of their sequence numbers. Forward compatibility is provided by ignoring message contents that is not expected at the receiving end. Any expected content that is missing however, raises a warning; either at message reception or during message flow configuration.

Variation in representation is enabled by message variant definitions of which examples are shown in Listings 6.5 and 6.6. Such variation is allowed to support differences in platform APIs and cultural preferences. It enables multiple variants of a single message type (and the same content) to exist with different data types and units for the contained data elements. Conversions between such variants of a message type are typically rather straightforward. They can be implemented within dedicated software components that can be inserted into the message flow. By means of a formal specification of conversion rules, such conversion components can be automatically generated upon binding two varying, yet compatible components. Such automated mediation however remains an item of future work.

6.3.4 Serialising messages and message definitions

TalkSens defines a lightweight serialisation format for messages and their run-time retrievable definitions. This format closely resembles the structure of MDL message definitions, but eliminates all elements that are not needed for run-time machine interpretation. To improve the development experience, the details of the serialisation format are hidden from developers by generated message-specific serialisation code (see Section 6.3.5).

To contribute to its compactness, serialisation uses numerical encoding and omits redundant human-readable naming. Message type identifiers are serialised in a variable size integer encoding, that omits leading zeros and supports large integers. Data types and units are encoded as shown in Table 6.1, and all serialisation is done in network byte order.

Figure 6.8 shows the message definition serialisation format. A serialised message type identifier is followed by a set of data records and elements. Serialisation of both elements and records starts with a *sequence number* (i.e. seqno in the figure) and a *flag*. The sequence number specifies which data element or record is being serialised. The flag indicates whether a data element or record is described and whether it contains a single value or an array. Data elements further specify the data type and unit of measurement used, while nested data records recursively specify the set of data elements they contain.

A serialised data message is a concatenation of serialised elements and/or records, shown in Figure 6.9, prepended by a message type identifier. Serialisation of elements and records is done in a *type-length-value* format. The sequence

```
                                                  +--------+--------+--------+--------+...
                                               ,' |            Data Element           |
                                             ,'   +--------+--------+--------+--------+...
                                           ,'     | seqno  |  flag  | d.type |  unit  |
+--------+----...----+--------+--------+,'       ,+--------+--------+--------+--------+...
|    Message type     | Data Record 0  |`.     ,'   1 byte   1 byte   1 byte   1 byte
+--------+----...----+--------+--------+  `. ,'
| #bytes | ID value  | seqno  |  flag  |   ,'.
+--------+----...----+--------+--------+,'    `+--------+--------+------------------------...
  1 byte    # bytes    1 byte   1 byte  `.     |            Data Record
                                          `.   +--------+--------+------------------------...
                                            `. | seqno  |  flag  |   Data Elements/Records
                                              `+--------+--------+------------------------...
                                                 1 byte   1 byte
```

Figure 6.8 – The message definition serialisation format.

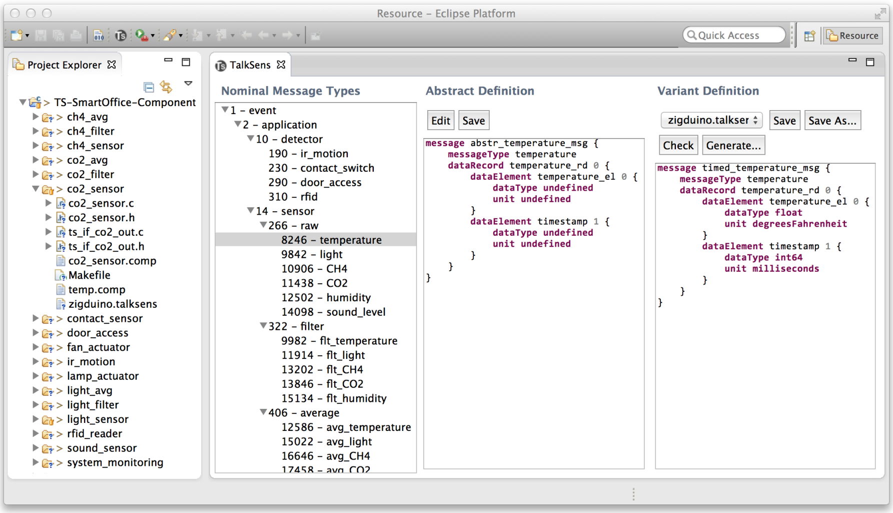
```
+--------+--------+-----...-----+     +--------+--------+------------------------...
|       Data Element            |     |           Data Record
+--------+--------+-----...-----+     +--------+--------+------------------------...
| seqno  | length |   value(s)  |     | seqno  | length | (serialised elements/records)
+--------+--------+-----...-----+     +--------+--------+------------------------...
  1 byte   1 byte   length bytes        1 byte   1 byte        length bytes
```

Figure 6.9 – The data element and record serialisation format.

numbers function as the type indicator and identify data elements and records within a message. The length indicator specifies the number of bytes that are needed to serialise the value at hand. For a data element, this depends on (i) the specific data type specified (See Table 6.1), (ii) whether it is a single or repeated data element, and (iii) on the value itself in case of a string. Data records, are internally serialised into string format and included as a string data element in their containing data record. Specific data types and units are not specified within the serialisation of data messages, as they are fully embedded within the generated serialisation code, as described in the following section.

6.3.5 Development support

The TalkSens framework includes an IDE (Integrated Development Environment) plugin to encourage the systematic use of previously defined messages during application development. As shown in Figure 6.10, the plugin allows developers to browse a central message definition repository and edit message definitions in the MDL format. Based on the chosen message definition, it furthermore allows to generate message-specific serialisation code in C or Java. This code can conveniently be included within the software project under development, where its API constraints the developer's interaction with the message payload and ensures correct writing and reading of contained values, respectively through mutator (i.e. *set*) and accessor (i.e. *get*) methods. The use



```
> TS-SmartOffice-Component
    > ch4_avg
    > ch4_filter
    > ch4_sensor
    > co2_avg
    > co2_filter
    > co2_sensor
        co2_sensor.c
        co2_sensor.h
        ts_if_co2_out.c
        ts_if_co2_out.h
        co2_sensor.comp
        Makefile
        temp.comp
        zigduino.talksens
    > contact_sensor
    > door_access
    > fan_actuator
    > ir_motion
    > lamp_actuator
    > light_avg
    > light_filter
    > light_sensor
    > rfid_reader
    > sound_sensor
    > system_monitoring
```

```
1 – event
  2 – application
    10 – detector
      190 – ir_motion
      230 – contact_switch
      290 – door_access
      310 – rfid
    14 – sensor
      266 – raw
        8246 – temperature
        9842 – light
        10906 – CH4
        11438 – CO2
        12502 – humidity
        14098 – sound_level
      322 – filter
        9982 – flt_temperature
        11914 – flt_light
        13202 – flt_CH4
        13846 – flt_CO2
        15134 – flt_humidity
      406 – average
        12586 – avg_temperature
        15022 – avg_light
        16646 – avg_CH4
        17458 – avg_CO2
```

```
message abstr_temperature_msg {
    messageType temperature
    dataRecord temperature_rd 0 {
        dataElement temperature_el 0 {
            dataType undefined
            unit undefined
        }
        dataElement timestamp 1 {
            dataType undefined
            unit undefined
        }
    }
}
```

```
message timed_temperature_msg {
    messageType temperature
    dataRecord temperature_rd 0 {
        dataElement temperature_el 0 {
            dataType float
            unit degreesFahrenheit
        }
        dataElement timestamp 1 {
            dataType int64
            unit milliseconds
        }
    }
}
```

Figure 6.10 – The TalkSens IDE plugin.

of such message-specific serialisation code is shown later in Listing 6.9 (Lines 45 - 46), and constrasts significantly with the ad-hoc approach in Listing 6.1.

An important design feature within TalkSens serialisation support is the split between message-specific and generic serialisation code. While the former is part of deployable application logic, the latter is included in the pre-deployed middleware. The generated serialisation code contains all, and only, message-specific serialisation logic. It only deals with a specific message's contents, data types and units of measurement. The additional middleware support complements this with generic functionality that allows to write/read a specified number of bytes to/from a message buffer. The result is that any data types and units of measurement that are not in use by deployed application logic, have no references within the software on a node. Consequentially, the TalkSens data model at the back-end can be extended with additional data types and units of measurement without the need to update any serialisation code already deployed within a sensor system. New application logic can however freely make use of those new primitives.

6.3.6 Implementation and integration with LooCI

The implementation of the TalkSens framework includes a library of tools to build and use the data model, the IDE plugin for development support, and an integration with LooCI.

The TalkSens library provides back-end support to instantiate a run-time version of the data model, and perform equality and compatibility checking of message definitions. The data model itself is externally specified using XML, which enables various instances of the TalkSens framework, that serve various sensor network deployments, to easily specify their own set of messages, data types, and units of measurement.

The TalkSens IDE-plugin is implemented for Eclipse Kepler and provides functionality as discussed earlier in Section 6.3.5.

The integration of TalkSens within LooCI, brings explicit event definition support to the latter. Instead of arbitrary lists of event types in source code, and tedious manual event serialisation, TalkSens brings more coordinated and less error-prone support for event definition and serialisation. In practice, the integration required small changes to component interfaces and the underlying middleware. The interfaces of LooCI components were adapted to make use of TalkSens-defined events, which includes both application of generated serialisation code, and support for the inspection of event definitions. The LooCI middleware was extended with generic TalkSens serialisation

```
1  /* TalkSens additions to the LooCI configuration API */
2  InteroperabilityReport ts_interoperable(EventType event, uint8
        srcComponentID, NodeAddress srcNode, uint8 dstComponentID, NodeAddress
        dstNode)
3  ResultCode ts_wire(EventType event, uint8 srcComponentID, NodeAddress
        srcNode, uint8 dstComponentID, NodeAddress dstNode)
4
5  /* TalkSens additions to the LooCI inspection API */
6  (ResultCode, uint8[] description) getProvidedInterfaceDescription(uint8
        componentID, EventType event, NodeAddress node)
7  (ResultCode, uint8[] description) getRequiredInterfaceDescription(uint8
        componentID, EventType event, NodeAddress node)
```

Listing 6.8 – Additional commands to the LooCI configuration and inspection APIs due to the integration of TalkSens support.

functionality, as well as underlying middleware support for the inspection of event definitions. LooCI's publish/subscribe event bus remains topic-based, with event dispatching based on event types only. However, additional TalkSens commands are added to the LooCI configuration and inspection APIs, discussed earlier in Section 4.3.3. The new commands are shown in Listing 6.8 and are used to check whether two interfaces (i.e. a provided and a required interface) are interoperable (Line 2), to wire two interfaces with an integrated interoperability check (Line 3), and to inspect interfaces for event descriptions (Lines 6 - 7). The integration was realised for two LooCI implementations; the C-based Contiki version for constrained sensor nodes like Zigduino and the Java-based OSGi version for more powerful mobile and back-end devices. These were selected to showcase TalkSens' cross-platform support, and its feasibility within constrained environments.

Listing 6.9 shows the TalkSens version of a LooCI/Contiki TemperatureSensor component, with TalkSens-specific lines of code highlighted in grey. For comparison, the plain LooCI version is shown in Listing A.6 in Appendix A. Most common code between both implementations is omitted and adaptations include the following. Two header files are included to respectively access the generic and message-specific serialisation support (Lines 6 - 7). The provided temperature interface is part of the component's state (Line 13) and must be initialised (Line 24) and closed (Line 29), respectively at the beginning and ending of the component's lifetime. Finally, by means of the message-specific serialisation code (Lines 41 - 43) a published event is intuitively constructed and published.

The resulting practical use of LooCI/TalkSens is depicted in Figure 6.11.

```c
#include "contiki.h"
#include "looci.h"
#include <stdint.h>
#include <avr/pgmspace.h>
#include "adc.h"
#include "talksens.h"
#include "ts_if_temperature_out.h"

#define TEMPERATURE_EVENT_TYPE 8246
#define ADC_CHANNEL 1

struct state{
   struct ts_interface* if_temperature_out;
   /* other state omitted */
};
static const struct state init_state PROGMEM = {/* ... */};

#define LOOCI_COMPONENT_NAME temperature_sensor
COMPONENT_NO_RECEPTACLES();
COMPONENT_INTERFACES(TEMPERATURE_EVENT_TYPE);
LOOCI_COMPONENT_INIT("temperature_sensor",struct state, &init_state);

static uint8_t init(struct state* compState, void* data){
   if_temperature_out_init(&(compState->if_temperature_out));
   return 1;
}

static uint8_t destroy(struct state* compState, void* data){
   ts_if_close(&(compState->if_temperature_out));
   return 1;
}

/* additional implementation code omitted */

static uint8_t time(struct state* compState, struct etimer* data){
   uint16_t adc_value = (uint16_t)readADC(ADC_CHANNEL);
   uint16_t temperature = (uint16_t)((adc_value * 167) / 1024) - 54;
   clock_time_t timestamp = (clock_time_t)clock_time();

   if(compState->if_temperature_out != NULL) {
      dr_temperature_out_set_timestamp(compState->if_temperature_out->
           data_record, timestamp);
      dr_temperature_out_set_temperature(compState->if_temperature_out->
           data_record, temperature);
      if_temperature_out_publish(compState->if_temperature_out);
   }
   /* additional implementation code omitted */
}
```

Listing 6.9 – The TemperatureSensor component implementation in LooCI/Contiki with TalkSens support. Lines of code that are specific for TalkSens are highlighted in grey.

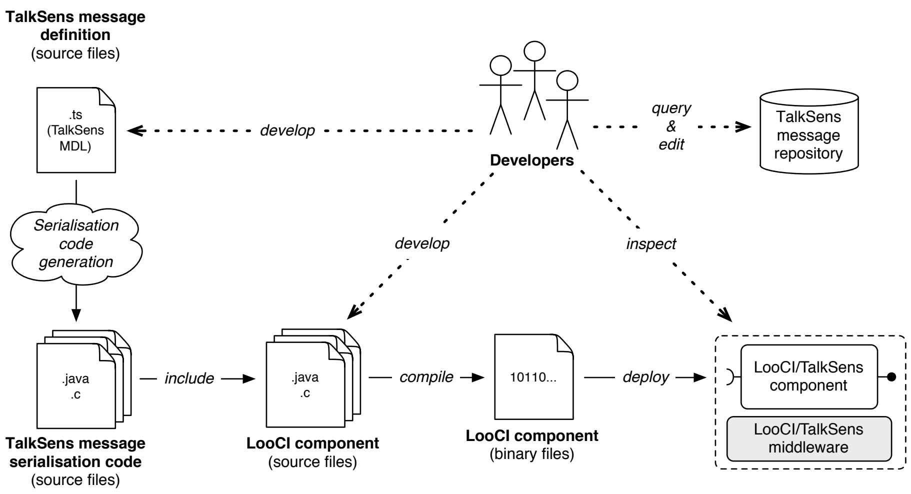

TalkSens message definition (source files)
.ts (TalkSens MDL)
develop
Developers
query & edit
TalkSens message repository
Serialisation code generation
develop
inspect
.java .c
include
.java .c
compile
10110...
deploy
LooCI/TalkSens component
LooCI/TalkSens middleware
TalkSens message serialisation code (source files)
LooCI component (source files)
LooCI component (binary files)

Figure 6.11 – Practical use of the TalkSens framework during LooCI component development.

Based on inspection of running components or querying of the message repository, developers decide which message (or event) definition to use in their LooCI component under development. From the TalkSens MDL description of the selected event, the corresponding serialisation code is generated for the platform of choice; i.e. OSGi or Contiki, and included into the component's source code. After compilation and deployment, the component can be wired to other components and inspected for event descriptions.

6.3.7 Related work

The WSN research literature does not report on similar integrated approaches to coordinated message definition as is provided by TalkSens. Yet, some alternatives for a number of its concepts have been presented.

The need for more formal interaction with explicitly specified structure of exchanged data has been identified by a number of recent efforts. These include open source initiatives like Project Haystack [149], and industrial consortium proposals like the IPSO SmartObject Guideline [79]. Both propose sets of message specifications that WSN integrators can decide to comply to, without any additional support. They apply a more strict data model than TalkSens, which either limits their use to specific applications like building monitoring [149], or prevents application-specific tailoring of messages [79].

Research efforts like SONGS [109] and Dioptase [14] provide development support for building distributed stream-based application compositions for sensor systems. They explicitly define the semantics and contents of data streams in a well-structured manner. SONGS, however, aims at building back-end applications that share sensor data. To this end, data that originates from various types of sensor devices in platform-specific formats is converted into an open and reusable format, such as XML. Dioptase, on the other hand, while including units of measurements into its data stream specifications, relies on JSON and (Web service) ontologies. By consequence, both efforts have been realised only for more powerful infrastructure, either gateways and servers [109] or less-resource constrained sensor nodes [14].

Within the scope of type-safe interactions, Lorien [145] argues with the use of opaque memory blocks with assumed internal structure for interactions between components in production-class environments. It provides a component-based modular operating environment that provides safety with strongly-typed interfaces by means of (i) formal interfaces for compile-time checking of type compliance, and (ii) hash-codes for composition-time (i.e. run-time) verification of type equivalence. This is a platform-specific and procedural solution, however, that only supports node-local interactions.

State-of-the-art serialisation support in sensor networks is provided for instance by Active Messages [21] for nesC/TinyOS, as shown earlier in Listing 6.2. Message definitions are specified using dedicated nesC-structs that offer serialisation support. Additionally, TinyOS comes with the *mig* (message interface generator) [121] tool that generates Java message objects from nesC packet descriptions. While this facilitates integration of TinyOS and Java applications, this solution is still platform-specific. Message definitions are still specified in source code and no additional coordination support for their use is provided. Dedicated serialisation within more powerful distributed systems is provided in a platform-independent manner by for instance Protocol Buffers [57] and Thrift [6]. Both provide a message definition language and generation of message-specific serialisation code for C++, Java, Python, etc. Although originally designed for back-end applications, more lightweight prototype implementations are also available; for instance nanoPB [128] for C and embedded devices, and a JavaNano version [81] for Android devices. These systems however do not deal with units of measurement, provide no light-weight run-time retrievable message descriptions, and include no coordination support to administer message definitions.

Finally, the explicit use of units of measurement in type systems has been investigated within the programming languages domain [87, 88, 175]. These efforts aim to guarantee scientifically correct calculations in terms of dimensions by incorporating units into language data types and the arithmetic applied on them. This extents TalkSens' aim to increase data interoperability. While such language support is currently not widely supported, nor available on sensor platforms, it would enable to enforce the correct use of units (see Section 6.3.3). Within sensor systems research, SpatialViews [129] mentions the use of units of measurement within the high-level programming language it provides to build distributed sensor applications. Also there, however, this remains an item of future work.

6.3.8 Conclusion

This section presented the second contribution of the TalkSens framework; coordinated definition of message types and contents. TalkSens thereto features a distributed architecture that provides development-time and run-time solutions to facilitate the use of (pre)defined messages. A message description language enables developers to define message contents in a platform-neutral manner, and an IDE plugin enables browsing of a collection of message definitions, and generating message-specific serialisation code that can be included in application components. Run-time support includes inspection of components for content definitions of the messages they exchange. This facilitates development of new components that can directly interact with running third-party components, or easy mediation between component interfaces that exchange semantically compatible messages with differences in content.

6.4 Discussion

This section reflects on TalkSens' subtyping and message definition support and places it within the larger scope of this dissertation.

First, while the proposed hierarchy encoding function was developed for subtyping, it can also be applied for other purposes. Two examples include (i) encoding CoAP resource paths, as presented in [171]; and (ii) SDlite status parameters, in contrast to the more naive and verbose method of encoding applied in Chapter 5. Furthermore, while this chapter primarily discussed the advantages of subtyping with regards to configuration, it also leads to increased flexibility during service discovery. Subtyping allows for a single query to be used to discover all components that feature an interface that is a subtype of the specified service (i.e. event type).

Second, the TalkSens message definition support improves run-time reuse of independently developed components. It does not, however, aim to establish global interoperability between software of fully independent stakeholders, as for instance initiatives like IPSO SmartObject Guideline [79] aspire to. Instead, it provides a framework in which a group of stakeholders can come to a semantic agreement of messages that are exchanged between their software components, yet, provide some flexibility in their contents for practical purposes; cultural differences, changing application requirements, and platform-specific issues in data representation. In addition with run-time retrievable message definitions from application components themselves, this improves the opportunities for reuse of application components across various compositions.

Third, TalkSens provides nominal subtyping of messages, but does not provide any inheritance support. Subtype relations are based only on the respective position of messages in a message hierarchy, and their resulting message type identifiers. Message contents is not inherited from supertypes, and only leaf types are expected to have their contents described within message definitions. Inheritance is omitted based on the practical consideration that only locally relevant message definitions and accompanying serialisation code should be stored on a node. Possibly unused supertype definitions and associated serialisation code is thus not pre-installed on sensor nodes. With this being the case, inheritance might only be of interest during the message definition phase to reduce the developer overhead; i.e. one message definition in the MDL format might automatically inherit the contents of the respective definitions of parent types. As this was not one of the primary research interests of the presented work, this was not further investigated.

Finally, the additional expressivity and flexibility in component bindings that is provided by TalkSens needs to be used in a responsible manner to avoid unwanted behaviour and misinterpretations during message exchange. For instance, in a dynamic setting in which new application components of multiple stakeholders are frequently deployed, abstract bindings should be created with caution. On the one hand, because existing bindings can affect the distribution of events related to those new components. This might be unwanted, and might easily go unnoticed by the new component's stakeholder. On the other hand, variations in message definitions are more likely to be present in these situations, which makes ensuring correctness of data exchange among all components involved in an abstract binding more complex. Consequentially, in such settings, one-to-one bindings of specific types are more favorable as possible compatibility issues can be detected by the TalkSens-specific LooCI wire command (see Listing 6.8). Nonetheless, subtyping can still be used during inspection and discovery.

6.5 Summary

This chapter presented TalkSens; a message definition framework for open sensor systems. TalkSens moves away from the ad-hoc and arbitrary definition of messages types and contents, as is common in many reported sensor system deployments. Instead, TalkSens promotes a more systematic and coordinated approach of defining messages as necessitated by more open use of application components in sensor systems.

Section 6.1 introduced the subtyping and message definition problems that TalkSens deals with. Section 6.2 presented an optimised hierarchy encoding function, which is based on the properties of prime numbers. The optimisation includes reuse of prime numbers across the hierarchy, and results in far more compact node identifiers. This enables the use of those identifiers and an accompanying subsumption test with integer representations and arithmetic that is standard available on sensor node platforms. The hierarchy encoding was applied within LooCI in support of event subtyping, which drastically reduces management overhead, as will be further shown in the evaluation in Chapter 7. Section 6.3 presented TalkSens' message definition support. It first discussed the state-of-the-art in the WSN domain and the short-comings thereof, and continued with presenting the TalkSens data model, development tools, and its integration with LooCI. The main benefit of TalkSens is that it facilitates interactions between independently developed application components and consequentially the reuse of these components across multiple compositions. Lastly, Section 6.4 reflected on the contributions of TalkSens and highlighted some attention points with regards to its practical application.

Chapter 7

Evaluation and practical application

This chapter evaluates the contributions presented in this dissertation and validates their applicability. All implementations of LooCI, SDlite, and TalkSens are quantitatively analysed in terms of performance, and a smart office deployment is used to evaluate their practical applicability.

In the following, Section 7.1 discusses the criteria against which the contributions are evaluated and Section 7.2 introduces the smart office use case. Section 7.3 provides a feasibility study of the integration of all contributions on constrained devices. Sections 7.4, 7.5, and 7.6 respectively evaluate LooCI, SDlite, and TalkSens in depth.

7.1 Evaluation criteria

In general, wireless sensor networks have hard restrictions on the available processing power, memory and bandwidth. Consumption of these resources furthermore has an impact on the available energy budget. To prolong sensor node lifetime, all the presented solutions therefore need to be lightweight in their computation, memory use, and communication. In relation thereto, the evaluation in this chapter primarily focuses on three criteria; middleware memory consumption, component sizes, and message sizes. These are important for the following reasons:

- **Middleware memory consumption.** The ROM and RAM consumption of the presented middleware solutions is of importance as together with the consumption of the operating system and other middleware services it determines the amount of memory that remains available for applications.
- **Component sizes.** The size of a component can be defined both as the amount of memory it consumes and the size of the deployable binary file that represents the component. Broadly speaking, the first determines the amount of components that can concurrently execute on a device, while the second determines whether a component can be dynamically deployed and the amount of energy this requires.
- **Message sizes.** As radio communication heavily impacts the energy budget within wireless sensor networks, any interaction should be minimised in terms of amount of data exchanged. When communication is needed, an effort should be made to prevent fragmentation and keep message sizes small enough to fit within a single radio packet's payload.

Additional evaluation within the chapter analyses the processing overhead and development effort of specific contributions.

7.2 The SmartOffice use-case

To evaluate the impact of the presented contributions in a practical setting, these were applied within a smart office deployment. Referred to as the *SmartOffice*, this has been installed within the offices of the DistriNet research group at KU Leuven. This section shortly describes its main features, and refers to Appendix C for a list of event descriptions, the event hierarchy, and an overview of the application composition. Additional details are presented in the doctoral dissertation of our colleague Javier del Cid [33].

The SmartOffice provides three services; (i) a comfort service that assesses the comfort level of office workers, (ii) a hazard service that detects fire, and (iii) a security service that detects unauthorised presence in the office and unattended open doors and windows. Additionally, on the non-functional level, network monitoring and data logging provides application status information and historic overviews of all gathered data. This is realised in a distributed setting in which application logic is deployed over various sensor system tiers; a sensor network for each office, a back-end server, and multiple clients. Figure 7.1 shows a simplified deployment diagram of the SmartOffice and illustrates where application logic for each service is deployed.

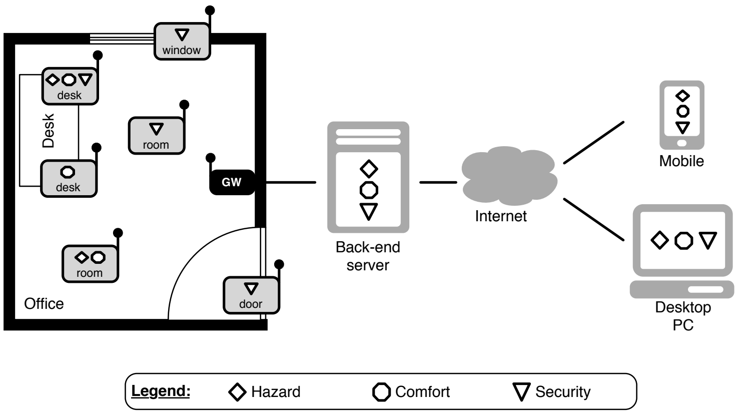


Figure 7.1 – Deployment diagram of the SmartOffice sensor system.

Each participating office is equipped with the following set of sensor nodes. Two nodes are installed in a central position within the office; the first one monitors the air quality (i.e. CO_2 and CH_4 levels), and the second one monitors the sound level and provides IR motion detection. Each desk within the office is also equipped with two nodes; one to monitor temperature and light, the other to activate a lamp and fan. These primarily relate to the environment of individual employees. Additionally, each window and door in the office is equipped with one node that uses a contact sensor to detect whether the window/door is open/closed, and in case of the door also detects whether a person enters/leaves the office. A typical office features six desks, two doors and two windows.

Besides LooCI components that detect these environmental conditions, additional components are deployed on the respective nodes that filter and average the temperature, light, CO_2 and CH_4 sensor readings before being forwarded to the back-end. A final component is deployed on each sensor node that provides the back-end with energy and node lifetime information.

At the back-end, the collected data is processed by hazard, comfort, security and network monitoring components and stored by a logging component. Two client application components are available to users to visualise the perceived comfort and potential security risks. These components are therefore subscribed to the back-end components that provide this data for each individual desk/user in an office.

	ROM	RAM	ROM	RAM	ROM	RAM
Hardware platform	**Zigduino-r1**		**Sun SPOT**		**Dell OptiPlex 780**	
Provided memory	128 kB	16 kB	4 MB	512 kB	250 GB	8 GB
Software platform	**Contiki**		**Sun SPOT**		**OSGi**	
Memory consumption	38,8 kB	9,9 kB	704 kB	47,7 kB	690 kB	321 kB
	(30%)	*(62%)*	*(17%)*	*(9%)*	*(<0,1%)*	*(<0,1%)*
LooCI	17,8 kB	1,6 kB	62,1 kB	32,0 kB	102,0 kB	11,3 kB
	(14%)	*(10%)*	*(1,5%)*	*(6,3%)*	*(<0,1%)*	*(<0,1%)*
SDlite	-	-	50,2 kB	1,8 kB	-	-
			(1,2%)	*(0,4%)*		
TalkSens	4,5 kB	12 B	-	-	3,2 kB	431 B
	(3,5%)	*(<0,1%)*			*(<0,1%)*	*(<0,1%)*

Table 7.1 – An overview of the memory footprint of the various LooCI, SDlite and TalkSens implementations on a heterogenous selection of platforms.

7.3 Feasibility study of the integrated approach

The combination of all contributions presented in this dissertation provides an application platform for open and multi-purpose sensor systems. This section evaluates the memory consumption of such an integrated effort for a number of hardware platforms that span the diverse sensor system tiers. More detailed evaluation for all of the contributions is provided in the following sections.

Table 7.1 lists the chosen hardware and software platforms and specifies the amount of ROM and RAM memory they respectively provide and consume. The Zigduino (revision 1) was selected as a constrained sensor node, the Sun SPOT as a more powerful sensor node, and the Dell OptiPlex 780 as a back-end server/client device. The respective underlying software platforms are the Contiki operating system, the Sun SPOT SDK and the OSGi framework. The rest of the table provides an overview of the memory consumption of the middleware contributions of this dissertation. This includes absolute values, and relative values in comparison to the amount of memory provided by each respective hardware platform.

With regards to LooCI, the specified memory consumption applies to the second version of LooCI.[1] For historic reasons, SDlite has only been implemented for, and integrated with, the LooCI-v1 version for Sun SPOT. On the other hand, TalkSens has been implemented for and integrated with the LooCI-v2 versions for Contiki and OSGi. From Table 7.1, it can be seen that all implementations

[1]In the rest of this chapter, unless stated otherwise, all references to LooCI imply its second version.

are small and consume little memory in comparison to the respective underlying software platforms.

It can thus be concluded that an integration of the application modularity of LooCI, the status-aware service discovery of SDlite, and the coordinated messaging approach of TalkSens is feasible and fits comfortably within the memory of the selected range of devices. Importantly, these lightweight implementations leave considerable memory available for the additional deployment of application components. For instance, in the worst case, presented by the Zigduino, an estimated guess is that a full integration of LooCI, SDlite and TalkSens would leave up to 40% of ROM and 25% of RAM available for application components. Evaluation in the following sections shows that this is sufficient to comfortably use sensor systems as multi-purpose infrastructure.

7.4 Quantitative evaluation of the application platform

As discussed in Chapter 4, LooCI provides two abstractions; individually deployable application components enable modular application building, and the distributed event bus allows for loosely-coupled interactions between those components. In support of these abstractions, LooCI provides a thin middleware layer that is pre-installed on all nodes in a LooCI sensor system.

This section evaluates the overhead of the LooCI abstractions in comparison with the bare underlying software platforms. First, the static overhead introduced by the LooCI middleware is evaluated in terms of memory consumption. Second, the more dynamic overhead introduced by deployable components and event subscriptions are evaluated in terms of memory consumption and lines-of-code, and processing overhead respectively.

7.4.1 Quantifying the static memory consumption of LooCI

The following paragraphs evaluate the memory footprint of the various LooCI implementations, and quantify the memory requirements of the LooCI middleware, bare LooCI components, and SmartOffice LooCI components.

		ROM (bytes)		RAM (bytes)		Binary (bytes)	Lines-of-Code LooCI	Lines-of-Code rest
LooCI/Contiki	(Zigduino-r1)	95	*(0,07%)*	35	*(0,21%)*	687	12	1
LooCI/SunSPOT	(Sun SPOT)	1800	*(0,04%)*	13669	*(2,6%)*	1800	5	3
LooCI/OSGi	(Dell OptiPlex 780)	1545	*(<0,001%)*	768	*(<0,001%)*	1545	9	10

Table 7.2 – Static memory consumption, binary file sizes, and a lines-of-code analysis of bare LooCI components for the various supported platforms.

LooCI middleware

The LooCI middleware enables the remote deployment and management of individual application components and implements the distributed event bus. It is pre-installed together with the underlying software platform on each node in a LooCI sensor system.

Both the ROM and RAM memory consumption of all LooCI middleware implementations were presented earlier in Table 7.1. As can be seen from the table, LooCI runs comfortably on all selected platforms. LooCI/Contiki introduces the largest relative memory overhead; i.e. 14% of provided ROM and 10% of provided RAM on the Zigduino-r1. As the other hardware platforms are more powerful, the respective implementations consume at least an order of magnitude less memory. Across all implementations, the LooCI middleware can thus be considered lightweight in terms of memory consumption.

Bare LooCI components

With regards to LooCI components, two size-related criteria are of importance; (i) their static memory footprint determines whether they can comfortably operate on top of the LooCI middleware and underlying software platform, and (ii) the size of their binary files represents the number of bytes that need to be transmitted during deployment of the component and impacts the energy budget of the sensor node and network.

To establish a baseline for these criteria, Table 7.2 presents the static memory consumption and binary file size of *bare* LooCI components; i.e. minimal functional LooCI components that do not include any application code. Both absolute and relative ROM and RAM consumption are presented. The relative values are in relation to the total ROM and RAM provided by the respective hardware platforms.

Across all implementations and platforms, the memory consumption of bare LooCI components is low; at the maximum less than one per mille of the available ROM, and 2,6% of the available RAM. Consequentially, application code can be encapsulated into individually manageable LooCI components with minimal additional memory overhead.

While the static memory footprint of a C-based LooCI/Contiki component is very low, the respective component binary file is an order of magnitude larger. This is caused by the ELF-file format that is used to encapsulate component code in support of dynamic linking. LooCI/Contiki, however, already provides an optimisation to the standard ELF overhead by making use of a custom ELF-format, called Compact ELF (CELF) [36], in which 32-bit data types are replaced by 16-bit or 8-bit data types. This optimisation reduces the binary file size from 1075 bytes to the 687 bytes presented in the table.

LooCI/SunSPOT components are compiled to jar-files, which are optimised and byte-code verified into a binary bundle prior to deployment. These bundles, called *suites* determine the amount of bytes to be transmitted during deployment, as well as the ROM consumption. While the latter is small, the RAM consumption of LooCI/SunSPOT components is considerably larger. This is caused by the Sun SPOT *Isolate* mechanism, which enables run-time deployment, and concurrent and isolated execution of multiple applications (i.e. components in the case of LooCI) on a node. This overhead is however acceptable for the memory-rich Sun SPOT nodes and only results in a RAM consumption of 2,6% for a bare application component.

LooCI/OSGi components are compiled to .jar-files, called *bundles*, and deployed as is to devices that operate in the back-end, such as desktop machines and servers. The sizes of these components are minimal in relation to the available memory resources and bandwidth.

The memory footprints specified in Table 7.2 imply that a large number of components can be deployed on individual sensor nodes. In practice, this is however typically constrained to a hand-full of components for technical reasons. Such reasons include the available memory to store component binary files, and the amount of dynamic memory a component is allowed to use. LooCI/Contiki, for instance, specifies a number of slots to which components can be deployed. The number of slots is a trade-off between the amount of components that can be deployed and the amount of memory available to each component. While the number of slots is configurable, by default it is set to eight. Where this should not be sufficient, either less memory can be attributed to individual components or additional nodes need to be deployed.

Component	**ROM** (bytes)	**RAM** (bytes)	**Binary** (bytes)	**Lines-of-Code** LooCI	rest
LooCI/Contiki					
temperature sensor	284	45	998	19	21
light sensor	285	45	1000	19	21
CO_2 sensor	283	45	996	19	21
CH_4 sensor	284	45	997	19	21
sound sensor	285	45	1000	19	22
IR motion sensor	556	63	1400	28	45
contact sensor	317	46	1071	19	27
door access sensor	438	60	1218	21	53
temperature filter	314	44	969	20	16
light filter	315	44	971	20	16
CO_2 filter	313	44	967	20	16
CH_4 filter	313	44	967	20	16
temperature averager	310	42	1012	19	21
light averager	311	42	1014	19	21
CO_2 averager	309	42	1010	19	21
CH_4 averager	309	42	1010	19	21
lamp actuator	162	36	794	19	23
fan actuator	161	36	792	19	23
system monitoring	735	61	1581	21	39
RFID reader	253	40	976	21	22
Average:	327	46	1037	20	24
LooCI/OSGi					
presence	1976	1228	1976	13	28
hazard	2248	1236	2248	14	52
comfort	2789	1218	2789	13	111
security	2498	1182	2498	13	72
network monitoring	2201	1252	2201	12	37
logging	1913	1239	1913	12	13
Average:	2271	1226	2271	13	52

Table 7.3 – Static memory consumption, binary file sizes, and a lines-of-code analysis of the SmartOffice LooCI components.

SmartOffice components

In addition to the baseline memory and binary file size evaluation of bare LooCI components, an evaluation of typical application components provides additional insight into the practical applicability of the LooCI component model. Therefore, Table 7.3 presents the memory footprint and binary file size of the SmartOffice application components. With averages of 327 bytes of ROM consumption and 35 bytes of RAM consumption for the LooCI/Contiki components, and 2270 bytes of ROM consumption and 1226 bytes of RAM consumption for the LooCI/OSGi components, confirm the small footprint of LooCI components on all platforms. The memory footprints remain well within scope of provided memory by the respective hardware platforms, and binary file sizes remain sufficiently small to be over-the-air deployable.

In conclusion, it can be stated that the added advantages of LooCI, i.e. application modularity, remote application management, and distributed interaction, are realised with acceptable memory consumption of both the middleware and components.

7.4.2 LooCI component development overhead

Component development within LooCI comprises (i) encapsulation of application logic into an individually deployable and configurable component, and (ii) adaptation of the application logic to the LooCI API for life-cycle management and event-based communication. While encapsulation requires additional code to explicitly define the externally accessible attributes of components (provided interfaces, required interfaces, properties, component name, etc.), adaptation to the LooCI API typically reduces the amount of code needed for life-cycle management and communication. For instance, only a single line of code is needed to publish an event (see Listings 4.6 and 4.7), which eliminates the otherwise needed code for e.g. socket management.

Table 7.2 presents a lines-of-code analysis of bare components for the various LooCI implementations. The *LooCI* lines-of-code represent the source code lines specifically related to LooCI, while the *rest* lines-of-code represent additional lines of code required by the underlying platform. The proportions of both vary as in the various implementations the LooCI API either wraps, and thus replaces, functionality of the underlying platform, or adds additional functions to convert native application modules into LooCI components. On average, developing an individually manageable bare LooCI component requires about 15 lines of code, of which 9 are LooCI specific.

Type of subscription	**size** (bytes)	**contents**
local	4	event type, src comp. id, dst comp. id
remote-from	5	event type, src comp. id, src node id, dst comp. id
remote-to	4	event type, src comp. id, dst node id

(event type = 2 bytes, comp. id = 1 byte, node id = 1 byte)

Table 7.4 – Memory consumption of subscription entries in LooCI/Contiki.

This number changes with the amount of provided interfaces, required interfaces, properties, etc. that are exposed for configuration and inspection. Table 7.3, however, indicates that for a real-world application, like the SmartOffice, the LooCI-specific lines-of-code remain fairly stable across a range of components. The *rest* lines-of-code in Table 7.3 include actual application code, in addition to lines of code required by the underlying platform.

In conclusion, LooCI component development does not negatively impact application development effort. While some effort is needed to convert opaque native application code into an individually manageable LooCI component, the application code itself is typically reduced by means of the communication and management abstractions provided by LooCI.

7.4.3 Performance of the event bus

The event bus abstraction of LooCI enables late binding of components. This section assesses the scalability of the event bus under a growing number of subscriptions, and evaluates the impact of the extra indirection it introduces. Important considerations in this regard are (i) the dynamic memory consumption per event subscription, and (ii) the delay introduced by the indirect communication between components.

Memory consumption of event subscriptions

The dynamic memory overhead of maintaining subscription tables can be expressed by the amount of information that is needed per subscription. This information is presented in Table 7.4 for the LooCI/Contiki implementation, which operates on the most constrained hardware platform supported, i.e. Zigduino. For all three types of subscriptions, only a handful of bytes is required. This low overhead is partially realised by locally maintaining mappings between

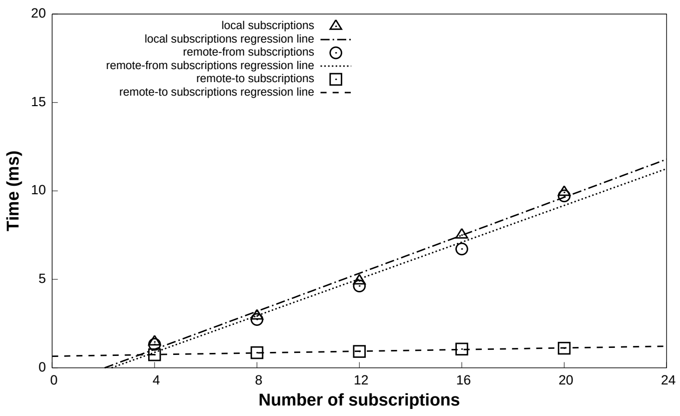


Figure 7.2 – Processing time required by the LooCI/Contiki implementation to dispatch events based on an increasing number of wires.

a 1-byte node identifier and the IPv6 address of nodes that participate in any remote subscriptions.

To evaluate the impact of this memory overhead, the presented numbers are applied to the SmartOffice sensor system. In that deployment, the node with the highest number of subscriptions contains 4 local subscriptions, 0 remote-from subscriptions, and 7 remote-to subscriptions. This results in a dynamic memory consumption of 44 bytes for the subscriptions, and 9 bytes for the IPv6 address mapping of the only remote node that is interacted with, i.e. the back-end.

Dispatch time of the event bus

The indirect style of communication provided by LooCI's publish/subscribe event bus, introduces a delay in component interactions. This delay is caused by (i) the evaluation of the current subscription entries, and (ii) the dispatching of events to all currently subscribed components. The graphs in Figure 7.2 visualise this delay for all three types of subscriptions; i.e. local subscriptions, remote-from subscriptions to dispatch events coming from the network to local components, and remote-to subscriptions to dispatch locally produced events to remote components. The graphs present a worst-case scenario in which all subscriptions are for the event type under consideration, and thus cause a copy of the event being dispatched. The time measurements were performed on a Zigduino-r1 clocked at 16MHz.

Event dispatching was timed between the moment the event enters the event manager and the moment its copy for the last subscription leaves the event manager; either towards a local component or towards the network stack. For local subscriptions this starts when the publishing component hands the event to the event manager and lasts until it is delivered to all subscribed local components. This includes retrieval of references to the destination component, copying the event, and passing the copy to the destination component's process. Similar measurements were taken for remote-from subscriptions, only there the event is passed to the event manager by the underlying network stack. For remote-to subscriptions, the measured interval lasts from the moment the publishing component passes the event to the event manager until it is passed to the underlying network stack. In this case, no hand-off reference needs to be dynamically retrieved, as their is only one network stack, and copying of the event occurs later when creating the respective event-wrapping UDP packet. Therefore the delay is considerably lower in the remote-to case. In terms of end-to-end communication also the transmission delay is of importance. This, however, depends greatly on underlying network protocols and their configuration and is therefore not considered here.

The presented graphs show that the delay caused by event dispatching grows linearly for an increasing number of subscriptions. Furthermore, the introduced delay remains below 10 ms for up to 20 subscriptions, which is almost twice the amount of subscriptions than needed for the sensor node with the most subscriptions in the SmartOffice deployment.

In conclusion, it can be stated that the performance of the distributed event bus scales well with a growing number of subscriptions, both in memory consumptions and added delay. A final remark to be made is that assuring this scalability is also a responsibility of the application manager. While the number of nodes in a network will typically increase the total amount of subscriptions in the network, it indicates bad design when this also directly impacts the amount of subscriptions per individual node.

7.4.4 Conclusion

LooCI's configurable abstractions for application modularity and distributed interactions make it a great enabler of multi-purpose sensor systems. The evaluation in this section shows that these features can be realised in a lightweight manner across all tiers of a sensor system, including resource constrained sensor nodes. LooCI's middleware and component memory consumption is sufficiently low to allow multiple application components to operate concurrently on a sensor node, and its event bus scales well in terms of added communication delay and memory consumption. Additionally, in terms of development effort, LooCI

	ROM		RAM	
Sun SPOT	**4 MB**		**512 kB**	
Sun SPOT VM	704 kB	*(17,2%)*	47,7 kB	*(9,3%)*
LooCI/SunSPOT	62,1 kB	*(1,5%)*	32,0 kB	*(6,3%)*
SDlite	**50,2 kB**	*(1,2%)*	**1,8 kB**	*(0,4%)*
Status registry	20,9 kB	*(0,5%)*	332 B	*(0,1%)*
Status provider	145 B	*(0,003%)*	17 B	*(0,003%)*
Service discovery	29,3 kB	*(0,7%)*	1,5 kB	*(0,3%)*

Table 7.5 – Detailed memory consumption overview of SDlite.

components are on par with native applications. In combination with LooCI's unified management APIs, this greatly facilitates development of distributed sensing applications across the various tiers of a sensor system.

7.5 Quantitative evaluation of the status-aware service discovery solution

As presented in Chapter 5, SDlite provides a status-aware service discovery solution that is tailored to sensor systems. It enables discovery of LooCI components with a specified provided or required interface on nodes that adhere to certain operational and environmental conditions. It therefor provides both a node-local registry that allows for sharing of status data across software modules and layers, and a service discovery protocol that makes use of this status data.

This section evaluates the status registry and service discovery solutions in terms of resource consumption and ease of use. It does so by quantifying (i) the middleware memory consumption, (ii) the programming and memory overhead for status sharing, (iii) the expressiveness of service queries in relation to message sizes, and (iv) the processing overhead of service query resolution.

7.5.1 Memory requirements of the SDlite middleware

The memory consumption of SDlite comprises that of the middleware support for both status sharing and service discovery, as well as the additional consumption by software components because of status sharing. The rest of this section discusses the middleware memory requirements, while the impact of status sharing is discussed in the following section. Table 7.5 provides an overview.

The first three lines in the table provide a reference and present the amounts of ROM and RAM memory that are available on a Sun SPOT, and used by the Sun SPOT VM and the LooCI/SunSPOT implementation. Besides absolute numbers, the table also presents the percentual memory consumption relative to the amount of ROM and RAM provided by the Sun SPOT hardware.

Both the status registry and the service discovery middleware are lightweight and together only consume a tiny fraction of the ROM and RAM available on the Sun SPOTs. This low memory consumption is primarily caused by a number of design decisions such as; (i) the integration of the service discovery mechanism with AODV and thereby leveraging its route discovery mechanism, and (ii) the design of the status registry, which simply provides references to where status parameters are locally available. A fully integrated implementation of both solutions, is furthermore believed to consume even less memory as some functionality is provided by both implementations; e.g. message parsing, request resolving, etc. This low consumption in memory leads to a firm belief that the same functionality can be implemented for even more constrained sensor node platforms than Sun SPOT.

7.5.2 Status provisioning overhead

To share status information, existing software components need to be transformed into status providers. This section discusses both the development overhead and the memory consumption of transforming an existing software component into a status provider.

In terms of development, to make a software service share status information it needs to implement and register the `IStatusProvider` interface. An example is shown in Listing 5.2 in Chapter 5. As shown in the listing, registration of a status provider requires a single line-of-code to call the `register()`-method. Additionally, to make the status parameters available to the registry, and its users, the `IStatusProvider` interface needs to be implemented. This interface provides a single method, `getStatus()`, of which the implementation needs to return the value of every status parameter shared. This is typically realised using a switch statement, resulting in about 2 lines-of-code per parameter. The programming overhead of status provisioning thus includes a fixed overhead of 3 lines of code, and increases linearly by 2 lines per shared parameter. Most status providers are furthermore expected to be middleware and system level software services; application developers will only rarely need to deal with status provisioning.

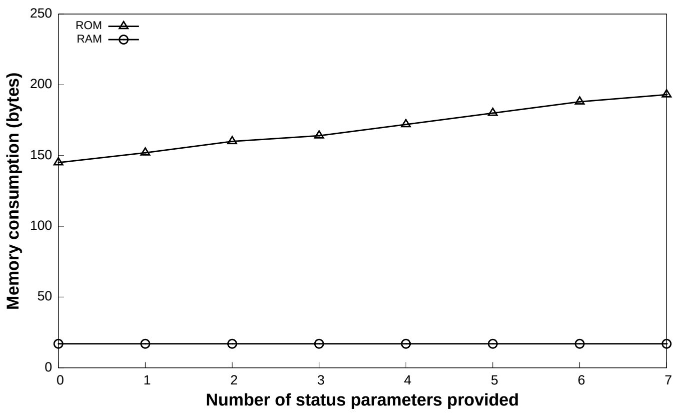


Figure 7.3 – ROM and RAM consumption of status parameter sharing.

As is to be expected, the impact of this added logic on memory consumption is low. Table 7.5 presents the memory consumption per status provider. Transforming an existing software component into a status provider, causes an average increase of 145 bytes in ROM consumption and 17 bytes in RAM consumption. While the ROM consumption can be attributed to the implementation of the `getStatus()` method (see Listing 5.2), the RAM consumption is caused by creating a new node in the tree data structure. This encompasses the status provider's encoded identification, a reference to the status provider itself, and a list of its child nodes. This is depicted as well in Figure 7.3, which additionally shows the average increase of memory use per status parameter provided. This is 8 bytes on average for ROM consumption, while RAM consumption remains constant. The latter is the result of the tree data structure only containing nodes for the status providers themselves and not for their status parameters. These parameters are already contained in the software service becoming a status provider and, by consequence, do not add additional dynamic memory overhead.

7.5.3 Evaluating service query expressiveness

Service querying enables the discovery of services (i.e. LooCI component interfaces) in a sensor network by disseminating a service request packet into the network (see Section 5.3.3). Besides the component interface type, the service query can include a status predicate that allows for the providing sensor node's environmental and operational status to be taken into account.

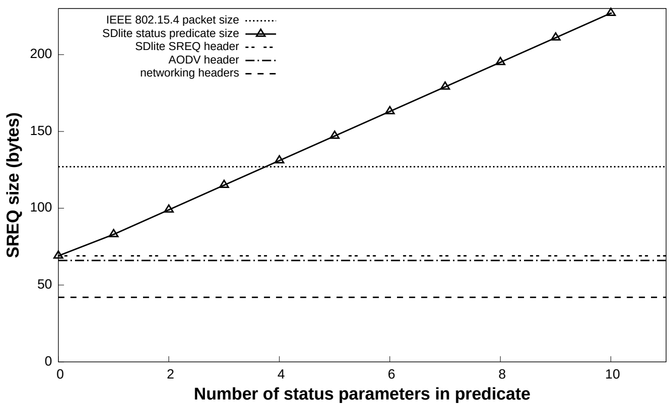


Figure 7.4 – Cumulative header and status predicate sizes in comparison to the total size of a single IEEE 802.15.4 packet.

The expressiveness of the service queries is demonstrated by the amount of status parameters that can be specified within such a query. While in theory SDLITE supports an infinite amount of status parameters, in the best case, a service request should fit into a single network packet, hereby avoiding fragmentation. Figure 7.4 presents the service request (SREQ) size in relation to the total available packet size within a Sun SPOT network[2]. The Sun SPOT platform utilises IEEE 802.15.4 which allows for radio packets with a total size of 127 bytes. Lower layer networking headers of the Sun SPOT networking stack consume 42 bytes. AODV headers, which are needed because of SDLITE's integration with AODV, consume an additional 24 bytes. On top of that, SDLITE adds a lightweight service request header of 3 bytes, which leaves 58 bytes for the status predicate.

The size of a status predicate[3] depends on (i) the amount of status parameters it contains, (ii) the size of their encodings, (iii) the data types and size of the values they are expected to have, and (iv) the amount of operators in the predicate. Figure 7.4 presents an impression of the total size of a status predicate for an increasing amount of status parameters and average sizes of parameter encodings and values. Based on the status parameter tree in Figure 5.2, status parameter encodings with a size of 3 bytes are applied as a representative average. Parameter values, on the other hand, can range in size from one byte (i.e. small number) up to tens of bytes (i.e. a string literal) per status parameter. An average of 4 bytes is therefore considered to be representative for practical

[2]A full description of the SDLITE packet formats is presented in Appendix B.
[3]A full description of the SDLITE status predicate format is presented in Appendix B.

purposes. Given these sizes, a status predicate with up to 3 parameters can be specified to fill the remaining payload of a single radio packet. Additional parameters can be specified as well, but will require fragmentation into multiple radio packets.

While SDLite thus provides an appropriate level of expressiveness at reasonable cost, regular fragmentation of service queries is expected within the presented implementation. Some small refactoring can however prevent this. On the one hand, more extensive compression of the status predicate can be achieved by applying varint encoding for values and the TalkSens sub-type encoding scheme to represent status parameter identifiers. On the other hand, in networks that use non-ad hoc routing protocols, like e.g. RPL [184], integration with routing has no added value. This frees up the 24 bytes needed for the AODV header.

7.5.4 Processing overhead of service query resolution

Service queries need to be resolved at all receiving sensor nodes, which should consume a minimal amount of processing to limit energy consumption. This section evaluates the time that is needed by the Sun SPOT implementation to retrieve a status parameter value and to resolve a status predicate. The first quantifies the additional processing caused by the extra indirection added by the central status registry for easy sharing. The second quantifies the time needed to parse the byte-array representation of a status predicate and the subsequent evaluation of that status predicate.

For an extended version of the parameter list shown in Figure 5.2, retrieving status parameters provided by middleware services takes on average 1,4 ms. When provided by LooCI components, this increases to 6 ms on average due to the inter-isolate communication between the middleware and system services, and the components. Variations in these timings are caused by the depth of the status parameter path within the status tree, the time needed to traverse the status tree to find the respective status provider, and possibly converting the retrieved value to a user-friendly format.

The time needed to resolve a status predicate depends on the amount of parameters it contains and the time needed to retrieve each of these parameters. As an indication, the average time to resolve the fairly complex predicate *((node/node_type = spot OR node/energy/vcc > 2500) & (node/memory/free > 200000))* was 87 ms. Thereof 80 ms was needed to parse the byte-array representation of the status predicate, including retrieval of the parameter values, and 7 ms for the stack-based evaluation of the reverse-polish predicate.

Both status parameter retrieval and status predicate evaluation remain within the lower milliseconds range, which is considered acceptable for practical purposes.

7.5.5 Conclusion

SDLITE allows for the run-time discovery of services within a sensor network based on descriptions of both functionality and sensor node status. Evaluation shows that both service discovery and status sharing can be implemented in a lightweight manner; SDLITE has a small memory footprint, status sharing only requires a few lines of code, and expressive service queries are resolved with low processing overhead. All these make SDLITE an extremely valuable and efficient tool to more open use of sensor networks.

7.6 Quantitative evaluation of the message definition framework

As presented in Chapter 6, the TALKSENS message definition framework provides support for event subtyping and explicit message definition. Section 7.6.1 evaluates the benefits of subtyping and quantifies at what cost this is achieved. Section 7.6.2 evaluates the impact of adding explicit message definition support to LOOCI.

7.6.1 Subtype encoding and its impact

The adapted subtype encoding function is quantitatively evaluated in the following manners. First, the extra compactness realised by reuse of prime numbers is investigated. Second, the processing cost of subsumption testing is quantified. And third, the impact of subtyping on the configuration of the SmartOffice deployment is discussed.

Compactness of subtype encoding with reuse of prime numbers

The adapted encoding function, presented in Section 6.2.2, reuses prime numbers during the process of encoding event type identifiers. This results into lower values of the latter, which can be represented using less number of bytes. This is an important improvement as it reduces the amount of bytes needed

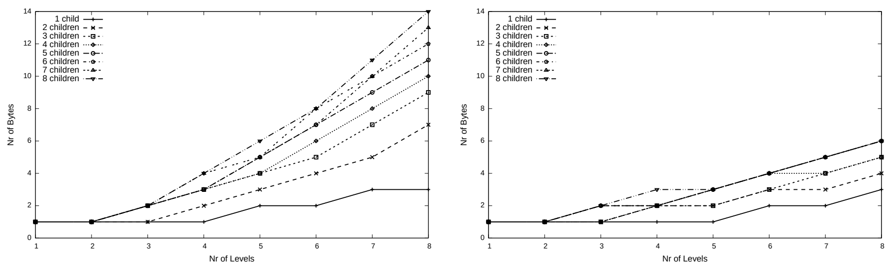
	Nb of vertices	Nb of primes	Largest id	Avg nb of bytes per id	Density of ids
Original	112	112	19.376.940	3	0,0006%
Adapted	112	24	30.914	2	0,36%

Table 7.6 – Comparison of the original and adapted encoding functions for the SmartOffice event hierarchy.

during communication, but also facilitates less complex subsumption testing on constrained sensor nodes.

Table 7.6 compares the original and adapted encoding function when applied to the SmartOffice event hierarchy (see Appendix C). This is a medium size event hierarchy that contains 112 event types. Consequentially, the original encoding function uses 112 prime numbers, while the adapted encoding function uses only about a fifth of those. This results in a much lower value of the largest event type identifier, which can be encoded using 3 bytes instead of 2 bytes for the original encoding function. Furthermore, the higher density, i.e. the percentage of integers used as identifiers up to the largest identifier, shows the more economic use of available integer values. While the byte-size reduction for the SmartOffice is limited, it does show that hierarchical information can be included within the event type identifiers in a very compact manner.

To evaluate the compactness of the adapted encoding function on larger event hierarchies, a set of artificial hierarchies were generated that range from 0 up to 8 levels, with each vertex having 0 up to 8 children. Figure 7.5 compares the original and adapted encoding function in terms of compaction; i.e. the number



(a) Original encoding function **(b)** Adapted encoding function

Figure 7.5 – Evaluation of the compaction of the original and adapted encoding functions for a set of differently shaped hierarchies.

of bytes needed to encode event type identifiers. The graphs show the byte-size of the largest identifier within the set of hierarchies.

The lower inclination of the graphs in Figure 7.5b clearly shows that the adapted encoding function results in more compact identifiers. While the original encoding requires up to 14 bytes, the reuse of prime numbers reduces this to only 6 bytes at the most. The adaptation thus keeps the byte-size well below 8 bytes for a wide range of event hierarchies; which can differ in shape (wide vs. narrow hierarchies, shallow vs. deep hierarchies) and size. As discussed in Section 6.2.2, this enables the use of standard integer representations on most (embedded) platforms, and eliminates the need for custom representations for large integers and accompanying arithmetic for subsumption testing. Practically, this means that event hierarchies can be comfortably designed without risking the requirement of such additional support.

Finally, it is important to highlight the trade-off between the total compaction of an event hierarchy and its overall shape. For instance, while reducing the number of levels within a hierarchy for the same amount of event types will lead to greater compaction, it will reduce the expressivity of the hierarchy; i.e. the flexibility of grouping various subsets of event types by one of their parent types. This trade-off should be well evaluated during design of the event hierarchy.

Performance evaluation of subsumption testing

Within LooCI, subsumption testing is performed to determine whether a published event needs to be dispatched according to an entry in one of the subscription tables. As this test is executed to evaluate every published event against all entries in the respective subscription tables, it needs to be performed with little processing overhead.

To quantify this overhead, an experiment was set up in which pair-wise subsumption testing was performed among all identifiers in the SmartOffice event type hierarchy. This evaluation was performed on Zigduino sensor nodes, which were clocked at 16MHz. The evaluated implementation of the subsumption test was optimised to first check whether the given parent identifier is not larger than the child identifier. If so, this rules out subsumption and the more expensive modulo operation does not need to be executed. The timing of this experiment showed an average of 19,7 µs per test. In comparison, pair-wise equality testing, as would be performed using a flat-list ordering, resulted in an average of 1,3 µs per test. Clearly, the subsumption test is a more complex operation to be performed in comparison to a simple equality test; yet, the required time for subsumption testing remains well within the acceptable lower microsecond range.

Location	**Number of wire commands**	
	Without subtyping	With subtyping
back-end	24	11
office	6 + (#rooms * 16)	2 + (#rooms * 8)
desk	8 + (#desks * 21)	4 + (#desks * 13)
window	1 + (#windows * 4)	1 + (#windows * 2)
door	1 + (#doors * 5)	1 + (#doors * 2)
client	#clients * 21	#clients * 7

Table 7.7 – Breakdown of the wire commands needed to configure the sensor nodes at various locations in the SmartOffice deployment.

Impact of subtyping on SmartOffice configuration

The goal of event subtyping within this dissertation is to reduce the management effort and accompanying dissemination of commands into a sensor network. As mentioned in Section 4.3.3, LooCI supports the late binding of, possibly remote, components by means of sending wire commands to the node(s) hosting those components. These commands result in respective entries in the various subscription tables of the targeted node's event manager. By using subtyping, individual subscription table entries can be used to dispatch multiple subtypes of events to a common destination. The end result is a reduction in management effort to define bindings, and in number of message transmissions to realise these bindings.

The reduction of wire commands is evaluated for the SmartOffice deployment. As discussed in Section 7.2, this deployment consists out of a back-end system that receives sensor readings from a set of sensor nodes monitoring desks, windows, doors and offices in general. After processing these sensor readings, comfort and security information is forwarded to a number of subscribed client applications.

Table 7.7 presents the amount of wire commands that are required to set up such a deployment; both without and with subtyping support. For each location within an office that is to be monitored, the amount of wire commands are presented; this amount consists out of a fixed base-line number of commands to prepare the back-end, and a variable number of commands per office, desk, window or door that is to be monitored.

For each monitored location, a clear reduction in wire commands is obtained through subtyping support. For instance, per sensor node that monitors a door, only a single wire command is needed to forward both the `contact_switch`

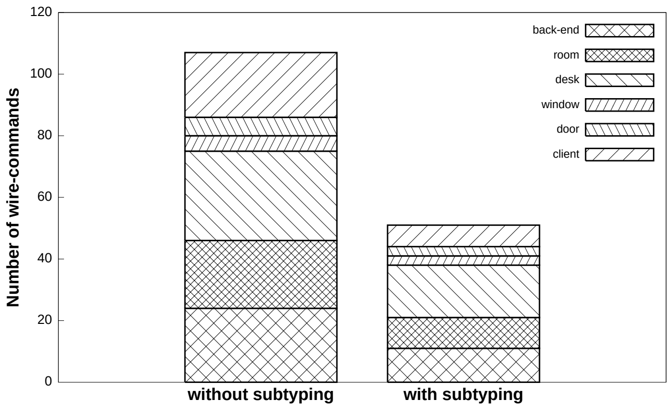


Figure 7.6 – The number of wire commands needed to establish communication in the SmartOffice deployment without and with subtyping support.

and `door_access` events to the security component in the back-end. In total, subtyping leads to a reduction from 107 to 51 in needed wire commands when configuring an example office with one desk, one window, one door and one client (see Figure 7.6). This corresponds to a substantial 52% reduction of respective messages that need to be disseminated into the sensor network. This reduction will further increase when additional offices, desks, windows and doors are added to the deployment.

As the sensor systems under scope in this dissertation are expected to be reconfigured over time due to changing application requirements, this reduction is additionally subject to a multiplication effect. On the other hand, this result is application specific and to a large extent depends on the structure of the event type hierarchy. However, sub-typing will never increase the number of binding commands required and when applied sensibly can reduce the amount of wire commands extensively. Additionally, similar savings can be expected when performing inspection or service discovery on a configured sensor system. In the SmartOffice for instance, a single service request for an abstract `actuator` service (i.e. event) can be used to discover all components that consume either `lamp` or `fan` actuation events.

7.6.2 Evaluating message definition overhead

As presented in Section 6.3, TalkSens makes message definitions more explicit at both development-time and run-time. Its purpose is to improve application reuse by facilitating interaction with third-party software. To determine the cost of this qualitative benefit, the following sections provide a quantitative evaluation of TalkSens' message definition support; specifically in terms of middleware memory use, deployable component sizes, and message sizes.

Part of the evaluation compares TalkSens serialisation support with Protocol Buffers [57]; a state-of-the-art serialisation mechanism that aims at being smaller and faster than XML. The functionality provided by TalkSens, however, does not fully cover that of Protocol Buffers, and vice versa. Compared to Protocol Buffers, TalkSens provides additional features such as message definition inspection support and specification of units of measurements. On the other hand, Protocol Buffers provides more extensive features to optimise and control message serialisation and exchange, which are not considered within TalkSens. Yet, the presented comparison indicates how the overhead of TalkSens relates to that of a closely-related state-of-the-art solution. In the following, the standard Java implementation of Protocol Buffers from Google [57], and the NanoPb [128] C-implementation are used, respectively to compare with the TalkSens implementations for LooCI/OSGi and LooCI/Contiki.

Memory overhead of the TalkSens middleware

TalkSens middleware functionality adds support for efficient message serialisation and inspection for message definitions to existing LooCI middleware implementations. A primary implementation concern was to realise this in a manner that is agnostic of the TalkSens data model (i.e. supported data types and units of measurement). This to prevent expensive middleware updates when changes to the data model occur. Table 7.8 presents the memory consumption of the TalkSens middleware in comparison to the respective LooCI implementations, and compares with Protocol Buffer alternatives.

	ROM		**RAM**	
LooCI/Contiki	17,8 kB		1,6 kB	
LooCI/Contiki - TalkSens	**22,2 kB**	*(+24,7%)*	**1,6 kB**	*(+0,1%)*
LooCI/Contiki - Protocol Buffers	29,6 kB	*(+66,3%)*	2,1 kB	*(+31,3%)*
LooCI/OSGi	102,0 kB		11,3 kB	
LooCI/OSGi - TalkSens	**105,2 kB**	*(+3%)*	**11,7 kB**	*(+4%)*
LooCI/OSGi - Protocol Buffers	596 kB	*(+484%)*	11,3 kB	*(+0%)*

Table 7.8 – Overview of the memory consumption of the TalkSens middleware and a comparison with related technologies.

In the case of LooCI/Contiki, TalkSens adds 4,4 kB of ROM consumption and virtually no RAM consumption. Both are well within the limits of the targeted constrained platforms and small when compared to the Protocol Buffers alternative. LooCI/OSGi-TalkSens performs even better with a 3% increase in ROM consumption and a 4% increase in RAM consumption. This is much smaller than the Protocol Buffers alternative. However, as mentioned, the Google Protocol Buffers implementation has more extensive serialisation support, and presumably was not implemented with memory constraints in mind.

Impact on component sizes

This section evaluates the combined effect on component sizes of (i) including message-specific serialisation code into LooCI components, and (ii) adding TalkSens inspection support to LooCI component interfaces.

Figures 7.7 and 7.8 provide an overview of the SmartOffice component sizes implemented for LooCI, LooCI-TalkSens and LooCI-Protocol Buffers, respectively for Contiki and OSGi.

In the Contiki case, component size refers to the size of an ELF-formatted run-time deployable component file. Figure 7.7 shows an average increase in component size of 1193 bytes by adding TalkSens support. Adding Protocol Buffer support, only results in a average increase of 411 bytes. The larger

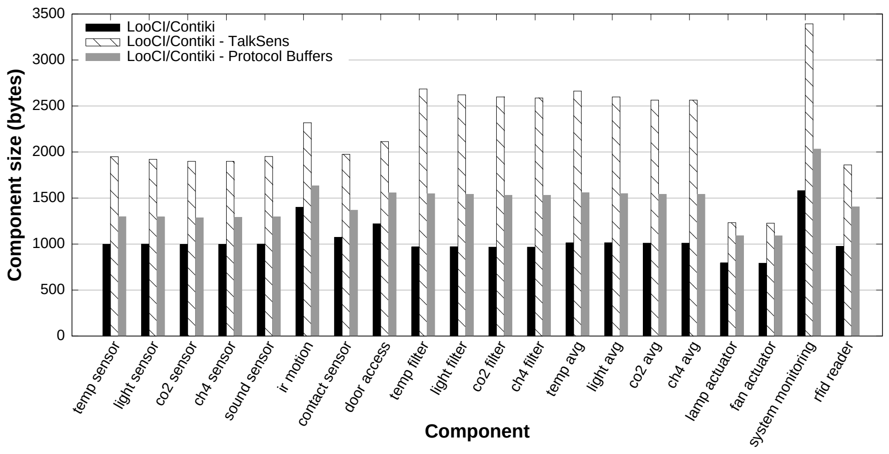


Figure 7.7 – Component sizes of SmartOffice Contiki components.

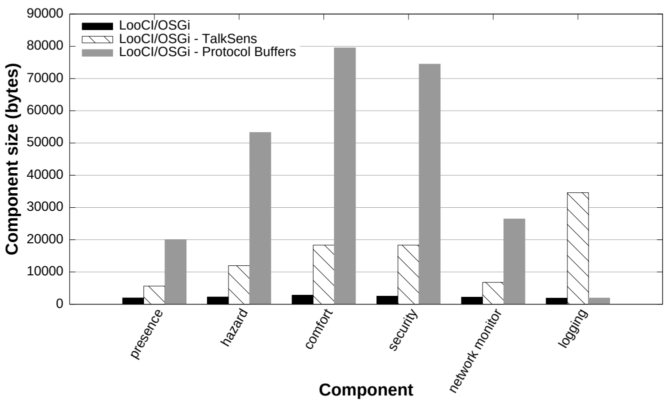


Figure 7.8 – Component sizes of SmartOffice OSGi components.

overhead of TalkSens can partially be attributed to the previously discussed reduced functionality offered by Protocol Buffers. Furthermore, TalkSens makes component interfaces a self-contained entity that provides serialisation and inspection of message definition. The implementation thereof is of the same order of complexity as the typical application functionality of a LooCI component; e.g. read out a sensor value and publish an event. Finally, it is important to notice that while for TalkSens the memory overhead is relatively larger for components, it is relatively smaller for the middleware (see Table 7.8). These proportions are reversed in case of Protocol Buffer support for LooCI/Contiki.

In the OSGi case, shown in Figure 7.8, component size refers to the size of a run-time deployable OSGi bundle in the form of a .jar-file. The figure shows an average increase in component size of 13,6 kB by adding TalkSens support. The relatively large increase of component sizes is expected as TalkSens interfaces introduce additional Java classes to LooCI components that otherwise typically consist of 2 classes only. Adding Protocol Buffer support, however results in a much larger average increase of 40,03 kB.[4]

To more objectively evaluate the component size growth introduced by TalkSens, Figure 7.9 shows the size of LooCI/Contiki and LooCI/OSGi components with an increasingly complex TalkSens interface. Complexity in this case is defined as the number of data elements in the interface's message, as

[4]The logging component forms an exception. It provides a generic interface and merely logs the byte-level representation of all incoming messages. Deserialisation is thus not strictly required for Protocol Buffers. In TalkSens however, by implementation choice, such support is fully integrated with a component interface and thus cannot be omitted.

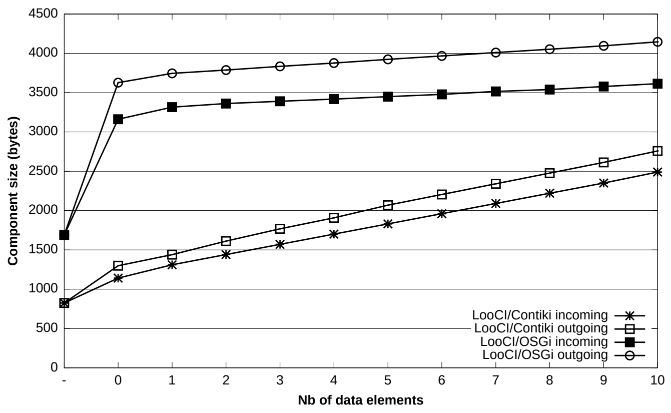


Figure 7.9 – Component size increase per added data element.

shown on the horizontal axis. The first data point on the graphs relates to bare components without interfaces. The next data points indicate the size of the same components with a respectively incoming or outgoing TalkSens interface without any data elements. This shows the boilerplate overhead of a TalkSens interface. With 475 bytes and 317 bytes on Contiki, and 1938 bytes and 1472 bytes on OSGi, respectively for outgoing and incoming interfaces, this is in agreement with the previous findings concerning the SmartOffice components. Further along the graphs, per data element the component sizes increase slightly with 146 bytes and 135 bytes on Contiki, and 45 bytes and 33 bytes on OSGi, again respectively for outgoing and incoming interfaces.

In conclusion, it can be stated that providing TalkSens support introduces a non-negligible overhead on component sizes on all platforms. Yet, in the Contiki case, components remain well within the scope of provided memory, allowing over-the-air component deployment and concurrent operation of multiple components on constrained sensor nodes. In the OSGi case, TalkSens components remain considerably smaller than their Protocol Buffer alternatives.

TalkSens data message and message definition sizes

To enable flexible serialisation, TalkSens adds meta-data to messages. This section evaluates the impact of the TalkSens serialisation format on message sizes and the size of serialised message definitions in support of inspection.

Figure 7.10 and Table 7.9 compare the payload sizes of the SmartOffice deployment's messages when serialised in three different formats; (i) a plain

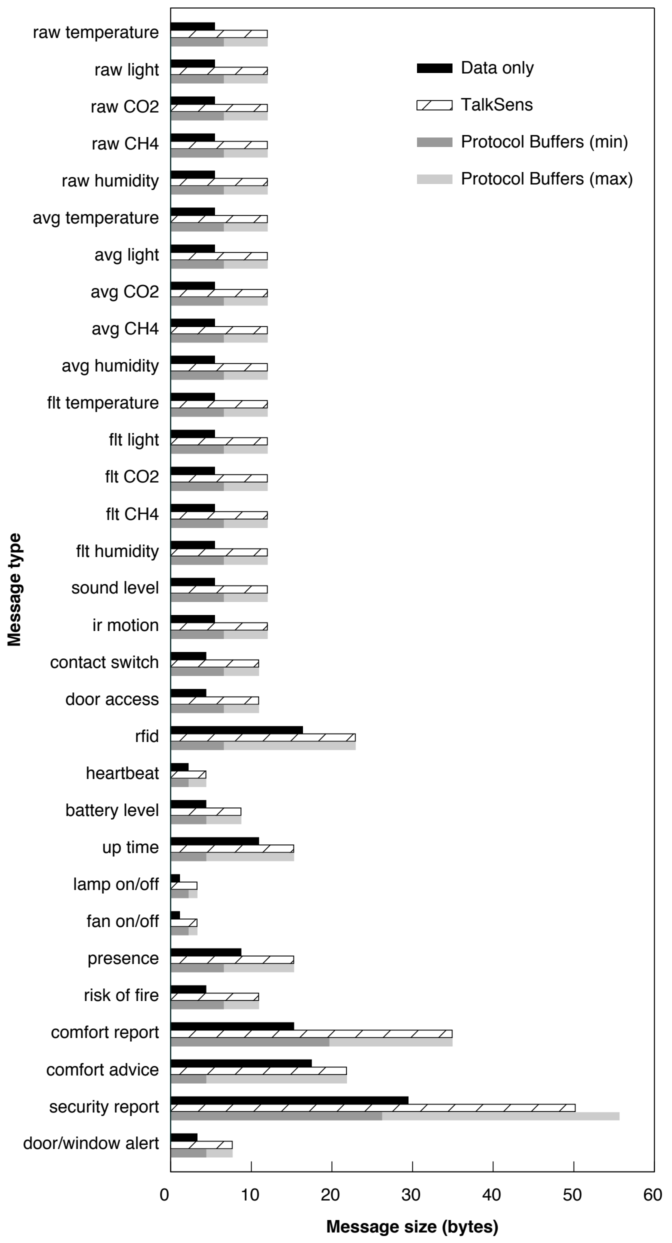


Figure 7.10 – TalkSens message sizes for the SmartOffice. (avg = averaged, flt = filtered)

Message types	**Data message size** (bytes)				**TalkSens description**
	Data only	**TalkSens**	**Protocol Buffers** (min)	(max)	
raw temperature	5	**11**	6	11	**17**
raw light	5	**11**	6	11	**17**
raw CO_2	5	**11**	6	11	**17**
raw CH_4	5	**11**	6	11	**17**
raw humidity	5	**11**	6	11	**17**
average temperature	5	**11**	6	11	**17**
average light	5	**11**	6	11	**17**
average CO_2	5	**11**	6	11	**17**
average CH_4	5	**11**	6	11	**17**
average humidity	5	**11**	6	11	**17**
filtered temperature	5	**11**	6	11	**17**
filtered light	5	**11**	6	11	**17**
filtered CO_2	5	**11**	6	11	**17**
filtered CH_4	5	**11**	6	11	**17**
filtered humidity	5	**11**	6	11	**17**
sound level	5	**11**	6	11	**17**
ir motion	5	**11**	6	11	**17**
contact switch	4	**10**	6	10	**17**
door access	4	**10**	6	10	**17**
rfid	15	**21**	6	21	**17**
heartbeat	2	**4**	2	4	**9**
battery level	4	**8**	4	8	**13**
up time	10	**14**	4	14	**13**
lamp on/off	1	**3**	2	3	**9**
fan on/off	1	**3**	2	3	**9**
presence	8	**14**	6	14	**17**
risk of fire	4	**10**	6	10	**17**
comfort report	14	**32**	18	32	**41**
comfort advice	16	**20**	4	20	**13**
security report	27	**46**	24	51	**37**
open door/window alert	3	**7**	4	7	**13**
Average:	6,4	**12,5**	6,3	12,7	**17,1**

Table 7.9 – Overview of sizes of data messages in various formats and message descriptions of the SmartOffice deployment.

data-only format, (ii) the TalkSens format and (iii) the Protocol Buffers format.

As previously shown in Figure 6.9, the TalkSens format introduces a 2-byte overhead for each data element/record within a message. This causes the SmartOffice message payloads to grow from an average size of 6,4 bytes in the data-only format, to about 12,5 bytes in the TalkSens format. Although doubling in size, this needs to be placed into perspective. Most messages, both in the SmartOffice and generally in sensor network deployments, contain only a few data elements that additionally are often only a few bytes in size. This causes the 2-byte overhead per data element/record to significantly influence the message sizes. Furthermore, the resulting message sizes remain well within the bounds of the available payload sizes in wireless sensor networks. In Contiki for instance, which operates on top of IEEE802.15.4, 6LoWPAN and IPv6, payloads of up to 86 bytes are allowed.

To account for Protocol Buffers' use of value-dependent varint encoding, a comparison is made against both its minimum and maximum payload sizes. Minimum payload sizes correspond with all values having a value of 0 or having zero length, while maximum payload sizes correspond with all values having their maximum value or a length that is representative within the application context. On average, the minimum and maximum payload sizes of SmartOffice messages in the Protocol Buffers format are between 6,3 and 12,7 bytes. This means that TalkSens serialisation is on par with Protocol Buffers maximum payload sizes. While this shows that additional compaction is possible, TalkSens does not require the more heavy-weight serialisation logic of Protocol Buffers, discussed in Section 7.6.2., This provides a good balance between serialisation complexity on the one hand, and message sizes on the other, which fits well with both the processing and energy constraints at hand.

Table 7.9 additionally presents the sizes of the SmartOffice serialised message definitions. The average size of 17,1 bytes shows that these are very compact and fit comfortably within the available message payloads.

7.6.3 Conclusion

TalkSens subtyping and message definition support respectively enable more expressive binding and improved reuse of application components. Evaluation of the adapted encoding function shows that it realises subtyping with a greater compaction of event identifiers and a less complex subsumption test than the original encoding function. The promised reduction in management overhead due to subtyping, is thus achieved with an additional reduction in communication and processing overhead. Second, evaluation of TalkSens' coordinated message definition support shows that its serialisation and inspection features require lightweight middleware support. Components and messages are, however, subject to a considerable increase in size, yet in typical cases, components remain over-the-air deployable and messages still fit within a single packet.

7.7 Summary

This chapter evaluated the contributions presented in this dissertation. In aid thereof, the SmartOffice deployment at DistriNet was introduced, which served as a practical application to evaluate their impact on a real-world deployment. Quantitative evaluation of LooCI, SDlite and TalkSens proved the feasibility of providing a run-time configurable application framework in support of open and multi-purpose sensor systems. Their combined middleware support is lightweight in terms of memory consumption and leaves sufficient memory available for applications on all selected platforms. Individually manageable and inspectable application components are small enough in size to be run-time deployable and allow for multiple components to execute concurrently on a single node. In terms of communication, the various messages defined by the contributions typically remain small enough to prevent fragmentation. These include application and management events, run-time retrievable event descriptions, and service discovery messages. Finally, event dispatching, subtype testing and service query resolution were shown to be performant on constrained sensor nodes. As such, run-time configurability, multi-purpose support, and extensive inspection and discovery of application logic are shown to be features that are feasible within the resource constraints of wireless sensor networks. This enables building distributed sensing applications that span the various tiers of a sensor system and support open use of third-party functionality. Yet, while considered feasible, all this does come at a cost. However, where application flexibility is of good use, this cost is considered to be well worth the investment.

Chapter 8

Conclusion

This chapter concludes the dissertation. Section 8.1 first summarises the presented contributions and highlights their main features. Section 8.2 then reflects on those contributions to highlight some important observations with regards to their use and applicability. Next, Section 8.3 identifies opportunities for future work. Lastly, Section 8.4 takes a step back and positions the presented work within an outlook on future sensor systems.

8.1 Contributions

Driven by the intent to improve the return-on-investment of sensor network infrastructure and development, this dissertation explored the feasibility of sensor networks as open and multi-purpose infrastructure. Such application-agnostic infrastructure can serve various stakeholders, who can independently deploy their applications and, where opportune, reuse the functionality provided by others. Several key features were identified to realise such a shared sensor network infrastructure. These include (i) the independent development and run-time management of individual modules of application logic, (ii) composition of such modules into distributed applications that span the various tiers of a sensor system, (iii) fine-grained discovery of application logic, and (iv) the availability of abstract descriptions of the messages exchanged by application modules. The contributions presented in this dissertation implement these features and together provide an application platform for multi-purpose sensor systems.

A related work study in Chapter 2 presented state-of-the-art solutions in support of open and multi-purpose sensor networks. Along with an overview of specific WSN technologies and the application of service-orientation within sensor networks, a survey of integrated application frameworks was presented. This showed that point solutions to specific problems are reported on, yet, to the best of our knowledge, no well-integrated solution exists that provides a high enough level of configurability and openness to result in truly multi-purpose sensor systems.

Chapter 3 introduced all contributions presented in this dissertation and illustrated how they collectively enable a service-oriented modus operandi in which multiple stakeholders make use of a shared sensor system infrastructure.

Chapter 4 presented important abstractions for application modularity and distributed interactions that facilitate the development and management of distributed applications. These abstractions are provided by LooCI, the Loosely-coupled Component Infrastructure, in the form of a run-time configurable component model and distributed event-bus. LooCI has been implemented on a range of hardware and software platforms, which enable developers to use the same set of abstractions to build applications that span the various tiers of a sensor system.

Chapter 5 presented a service discovery solution that takes the environmental and operational state of sensor nodes into account during the discovery of functionality. In support thereof, a status registry is provided that enables consistent and generic sharing of status data across software modules and layers on a sensor node. The integration of status sharing with service discovery results in fine-grained discovery of third-party application components within sensor networks.

Chapter 6 presented a solution for explicit and coordinated definition of messages to facilitate interactions with, possibly run-time discovered, third-party application components. The TalkSens message definition framework enables multiple parties to come to a shared agreement on message types and contents, and provides associated development and run-time support. Its subtyping support considerably reduces configuration effort when building distributed applications, and its message definition framework facilitates meaningful and error-free interactions between third-party application components.

Finally, Chapter 7 evaluated the prototype implementations of all presented contributions and their application within a smart office case-study. Evaluation showed that open and multi-purpose use of sensor systems can be established with acceptable cost in terms of memory, performance, communication, and development overhead.

8.2 Observations

When looking back over the presented research, a number of observations can be made on the use and applicability of open and multi-purpose sensor systems. This section shortly describes the most important ones.

A first observation is that application modularity and run-time configuration are extremely valuable within sensor networks. In our experience, these have not only proven their worth during the final deployment of distributed sensing applications, but during the entire lifecycle thereof. During initial development, for instance, incremental adaptations can be easily made to independent software modules without affecting other parts of a prototype system. This greatly increases development speed. Additionally, inspection and discovery support are extremely valuable tools to maintain a clear overview of the deployed applications and their configuration. These features are also of great value during final deployment of applications on operational sensor system infrastructure. Within sensor systems, the actual environment often has a significant influence on application behaviour. The ability to perform last-minute fine-grained reconfigurations, or even component deployments, is of great value. The cost that this flexibility inherently introduces largely disappears once the sensor system goes into stable operational use.

Second, while the presented research aims to improve the reuse of application logic within sensor systems, it still remains the responsibility of the developer to fully realise that potential. In relation to the presented solutions, this includes well-thought of (i) decomposition of distributed application logic into simple, yet generic, application components, (ii) arrangement of an event hierarchy that results in practical subtyping opportunities, and (iii) definition of application-independent and reusable message contents. Historically, attaining effective reuse has been shown to be a difficult task throughout the distributed systems domain [41]. Opportunities might however be presented by the more restricted nature of the WSN domain; i.e. the variety of application functionality of a sensor network is basically limited to sensing, processing and actuation. This reduced scope might proof to facilitate better reuse of application logic.

Third, the research presented in this dissertation predominantly focused on facilitating application development for open multi-purpose sensor systems. Other orthogonal concerns are however of great importance as well in such dynamic environments. A number of these have been the topic of investigation within our research group; (i) additional security support is needed to provide access-control and ensure system integrity [111], (ii) resource management is required to provide fair and controlled access to sensor node resources [33], (iii) high-level application management is needed that abstracts away the details exposed by the LooCI management API [68], (iv) application behaviour of components needs to be coordinated across various compositions [29], and (v) proper understanding is needed of the impact of dynamic component-based configuration on the energy budget of sensor networks [150]. The contributions of this dissertation therefore need to be integrated with these systems to provide complete support for open multi-purpose sensor networks within industrial settings.

And finally, the presented contributions only practically support the reuse of previously unknown third-party application logic where an acquaintance relationship exists between those parties. For instance, a-priori agreements must be made on the SDlite and TalkSens data models for various parties to be able to correctly discover and interact with each others application components. While in many use-cases this is not an immediate constraint, it does impose a non-trivial limitation on the applicability of the presented solutions. The following section elaborates on this observation with regards to future work.

In addition to these observations, a number of recurring ideas deserve a second thought and merit further research as they may be even more widely applicable.

One technique that has been repeatedly applied throughout the presented research, is *numerical encodings*. These have been used to represent event types in LooCI, status parameters, data types and operators in SDlite, and data types and units of measurement in TalkSens. The main reason thereto is to reduce communication, memory, and even processing overhead within the sensor network in comparison with alternative human-readable string representations. This approach exploits the fact that, most often, sensor nodes are not user-centric devices; e.g. they lack a visual display. Instead, they typically rely on the more resource-rich back-end for user interaction. This feature is leveraged to apply lightweight numerical encodings within the resource-constrained sensor networks and only resolve them into their human-readable counterparts where user interaction occurs; i.e. at the more resource-rich back-end. A similar approach is for instance taken by OMA LWM2M [134], which replaces string-based CoAP resource paths by encoded variants.

A related concept is the *append-only policy* that has been imposed on the SDlite status parameter tree, and TalkSens event hierarchy and data model. The primary purpose for this constraint is to deal with the concurrency in development and run-time operation exhibited by sensor systems. As there is no strict temporal boundary between the two, optimisations such as full re-encoding of data representations cannot be performed, as it would require the entire infrastructure to be reprogrammed to reflect the new encodings and prevent encoding inconsistencies. The append-only policy avoids such invalidation, and therefore trades-off the reduction in compactness with avoiding network-wide reprogramming. TalkSens, however, partially circumvents this trade-off by making its node-local middleware agnostic of its data model, hereby restricting data model references to within deployable components. Up to an extent, this allows the data model to evolve without requiring middleware updates.

8.3 Future work

This dissertation presented three important contributions towards the open and multi-purpose use of sensor systems. It however does not mark a complete accomplishment of that vision. Additional effort needs to be invested to further complete and improve the presented contributions, and adapt them to parallel evolutions within the general WSN domain. This section discusses five points of possible future work.

Large-scale validation. The presented research started from practical problems experienced during application development for distributed sensor systems. Extensive effort was invested to build performant and easily usable solutions, which were positively evaluated within a set of prototype sensor systems for smart environments and logistics. Additional validation of the presented contributions within larger-scale real-world settings will however be beneficial, as it is likely to provide greater insight in the dynamics and behaviour of such systems and the applicability of the presented solutions.

Standards adoption. Recent standardisation of network and application protocols has resulted in more open and interoperable sensor networks. As more robust implementations thereof become available, an integration of application modularity and dynamic management with those standards will be a valuable next step towards the open use of sensor systems. Concrete examples include replacing the proprietary LooCI event bus with CoAP [156], CoAP Observe [65] or MQTT-SN [162] alternatives, and integration of TalkSens with recent standardisation efforts for data representations, for instance OMA LWM2M [134], IPSO Smart Object guideline [79], or SenML [82].

Improved tool support. Distributed sensor systems remain complex setups that continue to require a lot of expertise in terms of embedded development, networking and distributed computing. To relieve developers of some of those complexities, additional development and management tools can be devised. For instance, an on-line compilation and deployment service can off-load platform-dependent compilation and deployment to a back-end service, and shield developers from a heterogeneous set of tool-chains that are required across various sensor system platforms. Other tool support might include graphical application management, which abstracts away platform-specific and low-level management commands. Dedicated IDE plugins can for instance provide such an improved developer experience.

In-network mediation. Where application components interact directly within a sensor system, TalkSens can detect message content variations. This can be leveraged to provide fully automated run-time mediation in which dedicated conversion components are generated, based on the inspected abstract message descriptions, and deployed where needed. This would require an extension of the data model with conversion rules, and a more formal approach to ensure type safe operation. Exploring the features and possibilities of, for instance, process calculus might show to be beneficial in this regard.

Remove acquaintance assumption. Additional research is needed to further improve the open use of third-party application logic within sensor networks with reduced prior agreements between the various parties involved. Based on the principles and solutions put forward within the Semantic Web [100] domain, more extensive descriptions of functionality, data, and non-functional concerns can be realised. This may ultimately lead to fully automated node-to-node interactions across organisational boundaries. Such integration of sensor networks with the Semantic Web has already been the topic of investigation, yet this is either restricted to back-end representations of sensor network functionality and data [24, 109, 187], or requires relatively powerful nodes [60]. In-network use of Semantic Web concepts under the present resource constraints remains a big challenge. Yet, the recent standardisation of application level networking protocols and the practice of off-loading complexity to the back-end can proof to be important enablers in this regard.

8.4 Outlook

Wireless sensor networks have shown great potential to improve many of the processes that we rely on in our daily lives. They enable us to extensively monitor the physical world around us and act upon the knowledge thus retrieved. Despite this large potential, current sensor network research finds itself in an interesting state of flux. Up to a few years ago, a lot of effort was spent to develop useful abstractions closely related to the specifics of sensor networks. While this resulted in novel and valuable solutions, the industrial adoption of sensor networks stalled and never really took off as envisaged in the research literature.

Lately, however, one of the most exciting and attention-grabbing developments within computer science is the Internet-of-Things (IoT). Also relying on embedded sensing technology and wireless communication, the IoT aims to augment everyday objects, named *Things*, and integrate them with the digital world and the Internet. It hereby benefits extensively from the ongoing standardisation efforts in networking and application protocols for constrained devices as triggered by WSN research.

Compared to sensor networks, the IoT however typically exhibits less complexity in terms of distribution of application logic; Things sense one or more environmental quantities and send this data in a direct manner to cloud-based applications. It are the latter that deal with most of the complexity within the system. Furthermore, current IoT applications are predominantly static in terms of software deployed on Things. Increased customer satisfaction is, however, likely to be a drive for more flexibility and customisation. One straight-forward evolution that can be envisaged, is more extensive distribution of application logic and increased direct interaction between Things. WSN research in general, and more specifically the contributions presented in this dissertation, can considerably contribute to that evolution.

Interestingly, more and more IoT applications are commercially available, and while arguably still perceived as technology gadgets, the increased insights they provide us with are impacting our daily habits. So, while the IoT momentum has shifted away considerable attention from the more complex in-network WSN problems, it will be interesting to see whether the innovations triggered by further IoT adoption can in turn spark a new interest into, and provide some much needed solutions for complex sensor network problems.

Appendix A

Component development in LooCI/Contiki and LooCI/OSGi

This appendix presents additional details on component development for LooCI/Contiki and LooCI/OSGi.

A.1 LooCI/Contiki

LooCI/Contiki components are implemented using the C-macro-based development style of Contiki for component definition, and an event-driven approach to implement additional component behaviour. In the following, first a number of generic code snippets for component definition and implementation are discussed. Afterwards, a full component implementation is provided for clarity and completeness.

Component definition starts by the definition of a component's state variables. Such explicit definition is needed to support multiple instantiation and is done by means of the C-struct shown in Listing A.1. Amongst other variables, a component's state constitutes its reconfigurable properties. Each component (i.e. codebase instance) holds a reference to its individual state struct instance stored in dynamic memory. Optionally, this state struct can be initialised with default values as shown in line 4. Default values of the state struct are stored

```
struct state{
  /* declare state variables */
};
static const struct state init_state PROGMEM = { /* state variable
    initialisation */};
```

Listing A.1 – Component state definition in LooCI/Contiki

```
#define LOOCI_NR_PROPERTIES 1
static const char property_name[] PROGMEM = "property_name";
LOOCI_PROPERTIES({/* property id */, /* property type */, offsetof(struct
    state, /* state variable */), /* size */, property_name});
```

Listing A.2 – Component property definition in LooCI/Contiki.

```
COMPONENT_NO_INTERFACES();
COMPONENT_INTERFACES( /* list of event types */ );
COMPONENT_NO_RECEPTACLES();
COMPONENT_RECEPTACLES( /* list of event types */ );
```

Listing A.3 – Definition of a component's provided and required interfaces in LooCI/Contiki.

```
LOOCI_COMPONENT("temperature_sensor", struct state);
LOOCI_COMPONENT_INIT("temperature_sensor", struct state, &init_state);
```

Listing A.4 – Component declaration in LooCI/Contiki

```
COMP_FUNCS_INIT         // THIS LINE MUST BE PRESENT
COMP_FUNC_INIT(function)
COMP_FUNC_DESTROY(function)
COMP_FUNC_ACTIVATE(function)
COMP_FUNC_DEACTIVATE(function)
COMP_FUNC_EVENT(function)
COMP_FUNC_SET_PROPERTY(function)
COMP_FUNC_PROPERTY_IS_SET(function)
COMP_FUNC_TIMER(function)
COMP_FUNCS_END(NULL)    // THIS LINE MUST BE PRESENT

uint8_t function(struct state comp_state, void data);
```

Listing A.5 – Mapping component behaviour to system events in LooCI/Contiki.

in program memory (cfr. `PROGMEM`) to reduce run-time memory overhead of components.

State variables can be explicitly exposed as component properties as shown in Listing A.2. This allows for external reconfiguration of the component's state variables. Based on the number of properties defined in line 1, and the property names defined as in line 2, the list of component properties is specified as in line 3. Each property is defined by an identifier, a data type and its name. Additional information contains a reference to the respective state variable and the expected size. Properties in LooCI can be of types: byte, short, int, long, string, and byte array.

Next, the component's provided and required interfaces need to be defined (respectively also named *interfaces* and *receptacles*). Listing A.3 presents two macros for interface and receptacle definition each. One of these needs to be used to declare the component's interfaces and receptacles; either no interfaces or receptacles are offered (lines 1 and 3), or a list of matching event type identifiers is provided (lines 2 and 4).

The actual component declaration in LooCI/Contiki occurs via one of the two macros shown in Listing A.4. Both specify the component's name and refer to the state struct of the respective component. In case default values should be given to any of the state variables, a reference to the initialising state struct variable must be passed as in line 2.

The functional behaviour of a component in LooCI/Contiki is implemented in an event-based approach. Therefor a list of possible system events is mapped to respective event-handler functions that can be implemented by the component. These mappings are specified via a number of macros, as shown in Listing A.5, which under the hood create an array of function pointers. Possible events include the initialisation, destruction, activation and deactivation of the component, the reception of a LooCI event, the update of a property (before and after), and an expiring timer. The generic signature of the event handler functions that the component needs to implement is presented in line 12. Each event-handler function takes the function name as specified in the mapping and passes the component's state struct and function-specific data as parameters. Depending on the actual event being handled, the data that is passed can be a pointer to an expired timer, a pointer to a LooCI event, etc.

A full component implementation is provided in Listing A.6, which represents the TemperatureSensor component described earlier in Section 4.3.4.

```c
#include "contiki.h"
#include "looci.h"
#include <stdint.h>
#include <avr/pgmspace.h>
#include "adc.h"

#define RAW_TEMPERATURE_EVENT 44
#define ADC_CHANNEL 1

struct state{
  struct etimer et;
  uint8_t sample_frequency;
};
static const struct state init_state PROGMEM = {.sample_frequency=10};

#define LOOCI_COMPONENT_NAME temperature_sensor

COMPONENT_NO_RECEPTACLES();
COMPONENT_INTERFACES(RAW_TEMPERATURE_EVENT);

#define LOOCI_NR_PROPERTIES 1
static const char sample_frequency_name[] PROGMEM = "sample_frequency";
LOOCI_PROPERTIES({1, DATATYPE_BYTE, offsetof(struct state, sample_frequency
    ), 1, sample_frequency_name});

LOOCI_COMPONENT_INIT("temperature_sensor", struct state, &init_state);

static uint8_t activate(struct state* comp_state, void* data){
  ETIMER_SET(&comp_state->et, CLOCK_SECOND * comp_state->sample_frequency);

  return 1;
}

static uint8_t time(struct state* comp_state, struct etimer* data){
  uint16_t adc_value = (uint16_t)readADC(ADC_CHANNEL);
  uint16_t temperature = (uint16_t)((adc_value * 167) / 1024) - 54;

  PUBLISH_EVENT(RAW_TEMPERATURE_EVENT, &temperature, sizeof(uint16_t));
  ETIMER_SET(&comp_state->et, CLOCK_SECOND * comp_state->sample_frequency);

  return 1;
}

COMP_FUNCS_INIT         // THIS LINE MUST BE PRESENT
COMP_FUNC_ACTIVATE(activate)
COMP_FUNC_TIMER(time)
COMP_FUNCS_END(NULL)    // THIS LINE MUST BE PRESENT
```

Listing A.6 – The TemperatureSensor component implementation in LooCI/Contiki.

A.2 LooCI/OSGi

Component development in LooCI/OSGi is very similar to LooCI/SunSPOT. Therefore, detailed code listings specifically for LooCI/OSGi are omitted from this text and only an implementation example of the TemperatureDisplay component (see Section 4.3.4) is provided.

As shown in Listings A.7 and A.8, one notable difference in LooCI/OSGi is the split of the component definition into an explicit codebase class and a component class. As shown in Listing A.7, the codebase class specifies the common elements of all codebase instances, such as the code base name (line 10) and lists of provided and required interfaces (lines 11 - 12). Additionally, the factory-method `createLoociComponent()` needs to be overridden that creates new instances of the associated component class (lines 15 - 18), shown in Listing A.8.

```
 1  package looci.osgi.app.temperatureDisplay;
 2
 3  import looci.osgi.serv.constants.EventTypes;
 4  import looci.osgi.serv.impl.LooCiCodebase;
 5  import looci.osgi.serv.interfaces.ILooCiComponent;
 6
 7  public class TemperatureDisplayCodebase extends LooCiCodebase {
 8
 9     public TemperatureDisplayCodebase() {
10        super("Temperature display",
11           new short[]{},                                          // provided
12           new short[]{EventTypes.FILTERED_TEMPERATURE_READING}); // required
13     }
14
15     @Override
16     protected ILooCiComponent createLooCiComponent() {
17        return new TemperatureDisplayComponent();
18     }
19  }
```

Listing A.7 – The TemperatureDisplay codebase implementation in LooCI/OSGi.

```java
package looci.osgi.app.temperatureDisplay;

import looci.osgi.serv.constants.EventTypes;
import looci.osgi.serv.impl.LoociComponent;

public class TemperatureDisplayComponent extends LoociComponent{

   public TemperatureDisplayComponent() {
      /* optional addition of properties */
   }

   @Override
   protected void componentStart(){
      /* logic at component activation */
   }

   @Override
   protected void componentStop(){
      /* logic at component deactivation */
   }

   @Override
   public void receive(short eventType, byte[] payload) {
      if(eventType == EventTypes.FILTERED_TEMPERATURE_READING){
         PayloadBuilder pb = new PayloadBuilder(payload);
         short value = pb.getShortAt(0);
         System.out.println("[TemperatureDisplay] The current temperature
              is " + value + " degrees Celsius.");
      }
   }
}
```

Listing A.8 – The TemperatureDisplay component implementation in LooCI/OSGi.

Appendix B

SDlite message formats

This appendix provides additional details of the message formats used in SDLITE.

Figure B.1 presents the service request format, with further details of the status predicate field presented in Figure B.2. A status predicate is formatted using reverse-polish notation in which operators follow, rather than precede, their operands. Within SDLITE, the operands consist out of a datatype - parameter - value triplet, in which the datatype element is needed to correctly parse the value. This triplet is followed by an operator element, which specifies the relational or logical conditions that should exist between the parameter's actual and specified value. This completes a single parameter description within the predicate; e.g. one that expresses `/node/memory/free` $>$ `3000`. Additional operator elements can furthermore be used to combine various parameter descriptions into a more complex status predicate; e.g. one that expresses `(/node/node_type` $=$ `spot & /node/memory/free` $>$ `3000)`. Finally, Figure B.3 presents the service reply format. Its status description field has a similar format as the status predicate field in a service request. The only exception is that the operators are omitted since the actual values of the parameters are presented.

```
 0                   1                   2
  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 |T|I|A|S|V| res |        service type           |
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
 .                                               .
 .               status predicate                .
 .                                               .
 +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+

  T (message type bit):  indicates whether the respective message is a request
                         (= 0) or a reply (= 1)
I (interface type bit):  indicates whether this request concerns a provided
                         interface (= 0) or a required interface (= 1)
          A (all-flag):  indicates whether only one (= 0) or all (= 1) services
                         that comply with the request are searched for in the
                         network
       S (strict-flag):  indicates whether all status parameters have to be
                         complied with (= 1), or whether locally non-available
                         status parameters can be ignored (= 0)
       V (values-flag):  indicates whether the actual values of the status
                         parameters specified in the status predicate need to
                         be included in the service reply
                   res:  reserved
          service type:  a 2-byte service/event type identifier
      status predicate:  a variable length, byte encoded status predicate in
                         reverse-polish notation
```

Figure B.1 – The SDLITE service request message format.

```
0                   1
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   DATATYPE    |   datatype    | ---,
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+    |
                                     |
        ,----------------------------'
        |      0                   1
        |       0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 ...
        |      +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...+-+-+
        '----- |   PARAMETER   |  param. size  | param. path | ---,
               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+...+-+-+    |
                                                                  |
                             ,------------------------------------'
                             |      0
                             |       0 1 2 3 4 5 6 7 ...
                             |      +-+-+-+-+-+-+-+-+-+-+-+...+-+-+-+
                             '----- |     VALUE     |     value     |
                                    +-+-+-+-+-+-+-+-+-+-+-+...+-+-+-+

                    0                   1
                     0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
                    |   OPERATOR    |   operator    |
                    +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure B.2 – The constituent elements of an SDLITE status predicate.

```
0                   1                   2
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|T|I|V|   res   |H|            TTL              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|        service type           |  nr of comps  |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
.                                               .
.                component list                 .
.                       +                       .
.              status description               .
.                                               .
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

T (message type bit):	indicates whether the respective message is a request (= 0) or a reply (= 1)
I (interface type bit):	indicates whether this request concerns a provided interface (= 0) or a required interface (= 1)
V (values-flag):	indicates whether this reply contains a status description field
res:	reserved
H:	indicates whether the time in the TTL field is specified in hours (= 1) rather than seconds (= 0)
TTL:	optional time-to-live of the service and status related information in support of caching
service type:	a 2-byte service/event type identifier
nr of comps:	the number of components specified in the component list field
component list:	list of 1-byte identifiers of components that satisfy the respective service request on the replying node
status description:	list of status parameters that were included in the status predicate of the respective request, and their actual values

Figure B.3 – The SDLITE service reply message format.

Appendix C

The SmartOffice sensor system

This appendix provides additional details with regards to the SmartOffice sensor system that was used throughout the evaluation in Chapter 7.

Section C.1 lists the events that are used within the SmartOffice and presents the respective event hierarchy. Section C.2 presents a detailed overview of the application composition.

C.1 Event list and hierarchy

The SmartOffice deployment specifies a set of LooCI event types that carry application data. Table C.1 lists these application event types and specifies their associated event type identifiers.

Alternatively, in support of subtyping, Figures C.1, C.2, and C.3 present the hierarchical ordering of these application event types and the LooCI management event types as used within the SmartOffice deployment. For each event type, the figures also specify the associated prime number and hierarchy-encoding event type identifier.

ID	**Event type**	**Contents**
1	raw temperature	uint16 timestamp, uint8 location_id, uint16 value
2	raw light	uint16 timestamp, uint8 location_id, uint16 value
3	raw CO_2	uint16 timestamp, uint8 location_id, uint16 value
4	raw CH_4	uint16 timestamp, uint8 location_id, uint16 value
5	raw humidity	uint16 timestamp, uint8 location_id, uint16 value
11	avg temperature	uint16 timestamp, uint8 location_id, uint16 value
12	avg light	uint16 timestamp, uint8 location_id, uint16 value
13	avg CO_2	uint16 timestamp, uint8 location_id, uint16 value
14	avg CH_4	uint16 timestamp, uint8 location_id, uint16 value
15	avg humidity	uint16 timestamp, uint8 location_id, uint16 value
21	flt temperature	uint16 timestamp, uint8 location_id, uint16 value
22	flt light	uint16 timestamp, uint8 location_id, uint16 value
23	flt CO_2	uint16 timestamp, uint8 location_id, uint16 value
24	flt CH_4	uint16 timestamp, uint8 location_id, uint16 value
25	flt humidity	uint16 timestamp, uint8 location_id, uint16 value
31	sound level	uint16 timestamp, uint8 location_id, uint16 value
32	ir motion	uint16 timestamp, uint8 location_id, uint16 timeout
33	contact switch	uint16 timestamp, uint8 switch_id, uint16 switch_state
34	door access	uint16 timestamp, uint8 switch_id, uint16 direction
41	rfid	uint16 timestamp, uint8 location_id, uint8[12] rfid_identifier
42	heartbeat	uint16 timestamp
43	battery level	uint16 timestamp, uint16 voltage_level
44	uptime	uint16 timestamp, uint64 uptime
51	lamp on/off	uint8 state
52	fan on/off	uint8 state
61	presence	uint16 timestamp, uint8 location_id, char[] user_name
62	risk of fire	uint16 timestamp, uint8 location_id, uint8 risk_level
63	comfort report	uint16 timestamp, uint8 location_id, uint16 temperature, uint16 light, uint16 CO_2, uint16 CH_4, uint8 fan, uint8 lamp, uint8 comfort_level
64	comfort advice	uint16 timestamp, char[] advice
65	security report	uint16 timestamp, uint8 risk_level, char[][] user_names, uint8[] door, uint8[] window, uint16 nb_people, uint8 movement, uint16 sound_level
66	open door/window alert	uint16 timestamp, uint8 switch_id

Table C.1 – Application events of the SmartOffice sensor system.

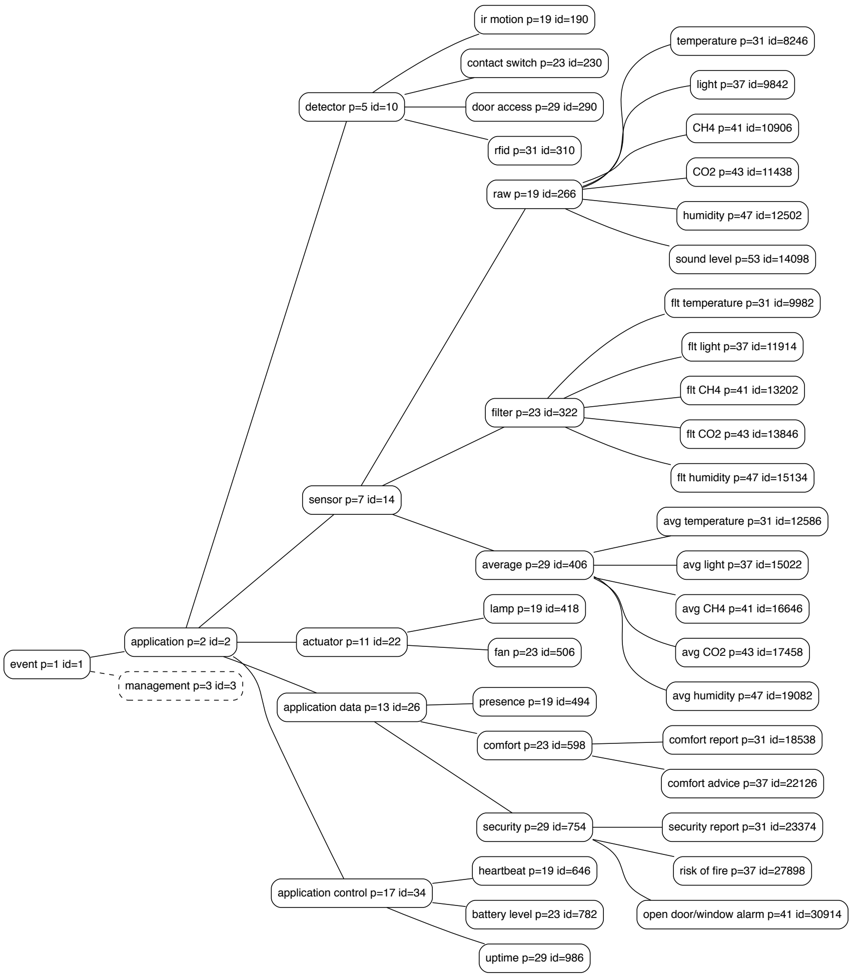


Figure C.1 – Partial SmartOffice event hierarch showing the application events. (Management events are further detailed in Figures C.2 and C.3.)

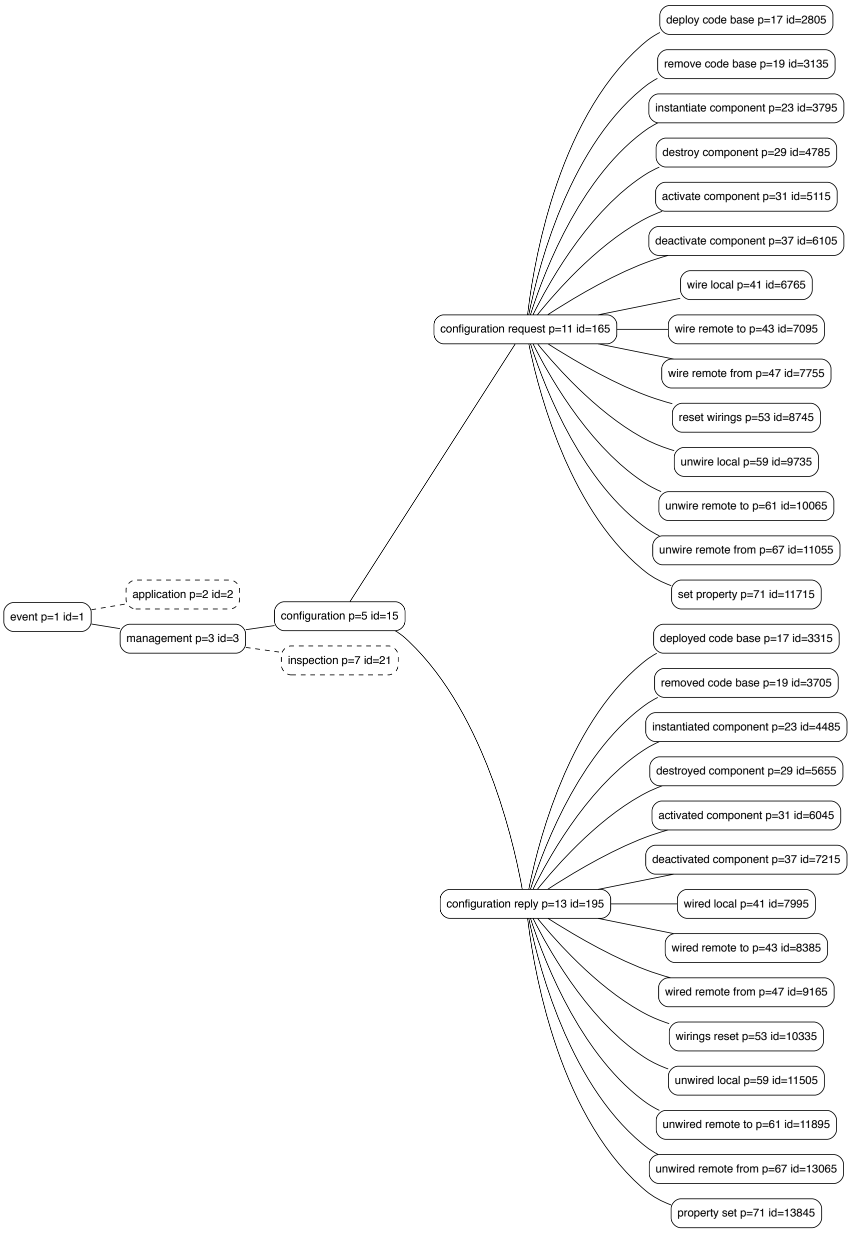

event p=1 id=1
application p=2 id=2
management p=3 id=3
configuration p=5 id=15
inspection p=7 id=21
configuration request p=11 id=165
deploy code base p=17 id=2805
remove code base p=19 id=3135
instantiate component p=23 id=3795
destroy component p=29 id=4785
activate component p=31 id=5115
deactivate component p=37 id=6105
wire local p=41 id=6765
wire remote to p=43 id=7095
wire remote from p=47 id=7755
reset wirings p=53 id=8745
unwire local p=59 id=9735
unwire remote to p=61 id=10065
unwire remote from p=67 id=11055
set property p=71 id=11715
configuration reply p=13 id=195
deployed code base p=17 id=3315
removed code base p=19 id=3705
instantiated component p=23 id=4485
destroyed component p=29 id=5655
activated component p=31 id=6045
deactivated component p=37 id=7215
wired local p=41 id=7995
wired remote to p=43 id=8385
wired remote from p=47 id=9165
wirings reset p=53 id=10335
unwired local p=59 id=11505
unwired remote to p=61 id=11895
unwired remote from p=67 id=13065
property set p=71 id=13845

Figure C.2 – Partial SmartOffice event hierarch showing the configuration events. (Application and inspection events are further detailed in Figures C.1 and C.3 respectively.)

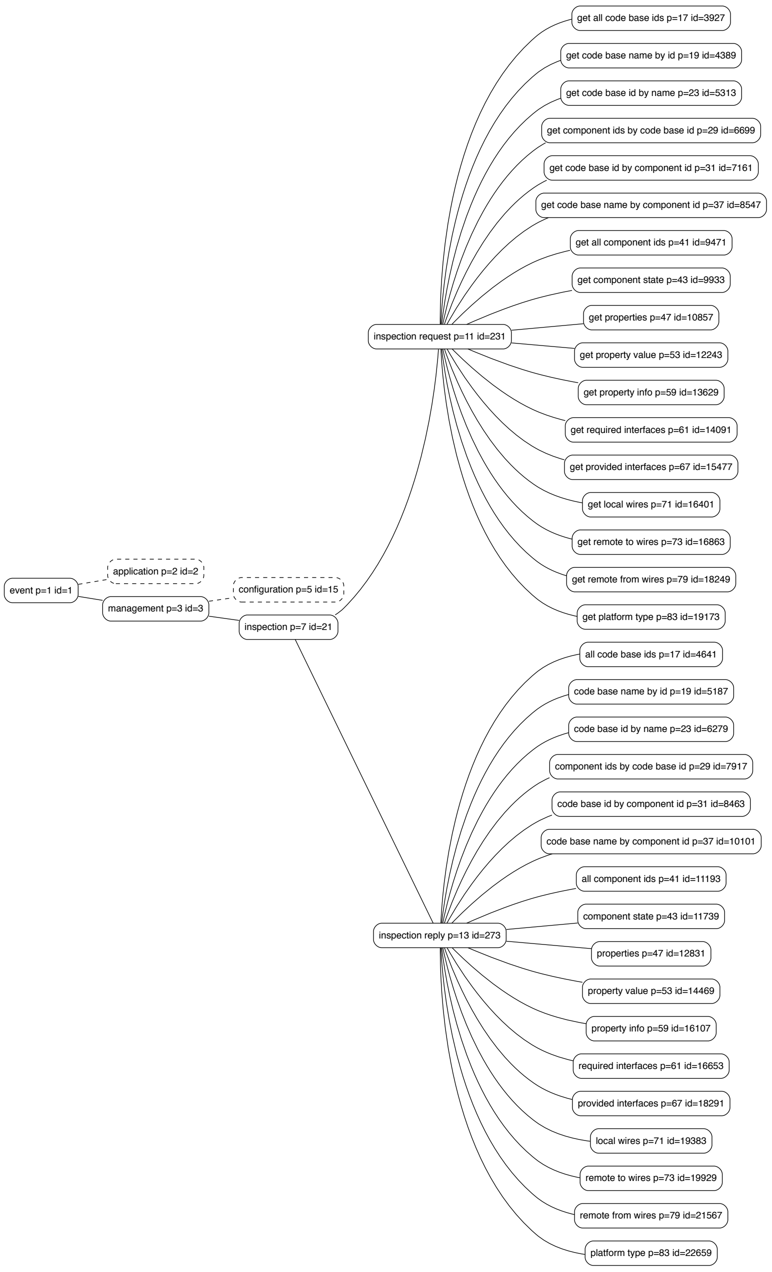

event p=1 id=1
application p=2 id=2
management p=3 id=3
configuration p=5 id=15
inspection p=7 id=21
inspection request p=11 id=231
get all code base ids p=17 id=3927
get code base name by id p=19 id=4389
get code base id by name p=23 id=5313
get component ids by code base id p=29 id=6699
get code base id by component id p=31 id=7161
get code base name by component id p=37 id=8547
get all component ids p=41 id=9471
get component state p=43 id=9933
get properties p=47 id=10857
get property value p=53 id=12243
get property info p=59 id=13629
get required interfaces p=61 id=14091
get provided interfaces p=67 id=15477
get local wires p=71 id=16401
get remote to wires p=73 id=16863
get remote from wires p=79 id=18249
get platform type p=83 id=19173
inspection reply p=13 id=273
all code base ids p=17 id=4641
code base name by id p=19 id=5187
code base id by name p=23 id=6279
component ids by code base id p=29 id=7917
code base id by component id p=31 id=8463
code base name by component id p=37 id=10101
all component ids p=41 id=11193
component state p=43 id=11739
properties p=47 id=12831
property value p=53 id=14469
property info p=59 id=16107
required interfaces p=61 id=16653
provided interfaces p=67 id=18291
local wires p=71 id=19383
remote to wires p=73 id=19929
remote from wires p=79 id=21567
platform type p=83 id=22659

Figure C.3 – Partial SmartOffice event hierarch showing the inspection events. (Application and configuration events are further detailed in Figures C.1 and C.2 respectively.)

C.2 Application compositions

The SmartOffice deployment features a set of sensor nodes that monitor offices, a back-end server and a set of client devices. The sets of components deployed on each node and the application compositions are presented in Figures C.4, C.5, C.6, C.7, C.8, C.9.

Notes:

1. Network monitoring, as shown in Figure C.9, is performed at each sensor node in the deployment.
2. All events that arrive at the back-end server or are locally produced there, are logged by a logging component that is deployed on that server. This logging component and the respective compositions are omitted from the figures for clarity purposes.

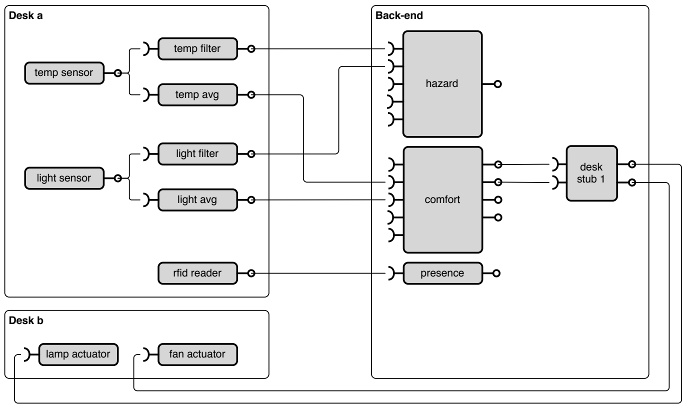


Figure C.4 – Partial application composition diagram of the SmartOffice deployment showing the LooCI wires between components on the desk monitoring sensor nodes and the back-end.

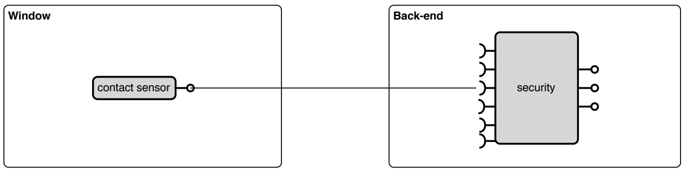


Figure C.5 – Partial application composition diagram of the SmartOffice deployment showing the LooCI wires between components on the window monitoring sensor nodes and the back-end.

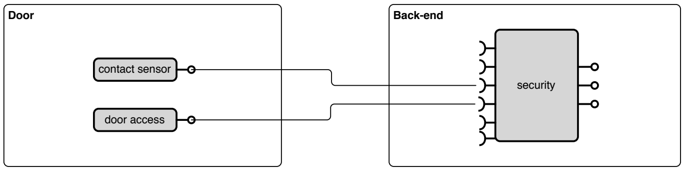


Figure C.6 – Partial application composition diagram of the SmartOffice deployment showing the LooCI wires between components on the door monitoring sensor nodes and the back-end.

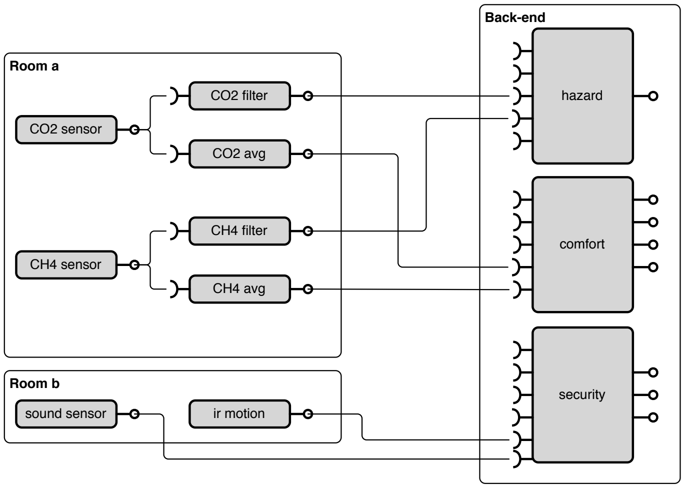


Figure C.7 – Partial application composition diagram of the SmartOffice deployment showing the LooCI wires between components on the office monitoring sensor nodes and the back-end.

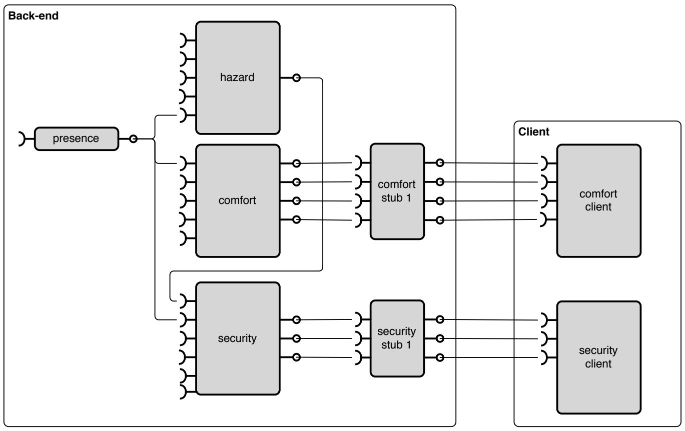


Figure C.8 – Partial application composition diagram of the SmartOffice deployment showing the LooCI wires between components on the back-end and client devices.

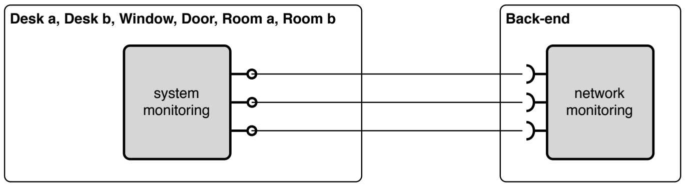


Figure C.9 – Partial application composition diagram of the SmartOffice deployment showing the system monitoring wires between all sensor nodes and the back-end.

Bibliography

[1] K. Aberer, M. Hauswirth, and A. Salehi. A middleware for fast and flexible sensor network deployment. *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1199–1202, 2006.

[2] K. Akkaya and M. Younis. A survey on routing protocols for wireless sensor networks. *Ad Hoc Networks*, 3(3):325–349, may 2005.

[3] J. N. Al-Karaki and A. E. Kamal. Routing Techniques in Wireless Sensor Networks: a Survey. *IEEE Wireless Communications*, (December):6–28, 2004.

[4] H. Alemdar and C. Ersoy. Wireless sensor networks for healthcare: A survey. *Computer Networks*, 54(15):2688–2710, 2010.

[5] F. M. Anwar, S.-w. Yoo, and K.-h. Kim. Survey on service discovery for Wireless Sensor Networks. In *Second International Conference on Ubiquitous and Future Networks (ICUFN)*, pages 17–21, Jeju Island, South-Korea, 2010. IEEE.

[6] Apache Thrift. `https://thrift.apache.org/`. (visited July 2015).

[7] K. Arnold, R. Scheifler, J. Waldo, B. O'Sullivan, and A. Wollrath. *Jini Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1999.

[8] Array of Things. `http://arrayofthings.github.io/`. (visited October 2015).

[9] E. Arseneau, R. Goldman, A. Poursohi, R. Smith, and J. Daniels. Simplifying the development of sensor applications. In *Object-Oriented Programming Systems, Languages and Applications (OOPSLA '06)*, 2006.

[10] R. Balani, C.-C. Han, R. K. Rengaswamy, I. Tsigkogiannis, and M. Srivastava. Multi-level software reconfiguration for sensor networks. In

Proceedings of the 6th ACM & IEEE International conference on Embedded software (EMSOFT '06), pages 112–121, Seoul, Korea, 2006. ACM.

[11] E. Bertino and R. Sandhu. Database security - concepts, approaches, and challenges. *IEEE Transactions on Dependable and Secure Computing*, 2(1):2–19, 2005.

[12] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms. *Mobile Networks and Applications*, 10(August):563–579, 2005.

[13] D. Bijwaard and W. van Kleunen. Industry: using dynamic WSNs in smart logistics for fruits and pharmacy. In *Proceedings of the 9th ACM Conference on Embedded Networked Sensor Systems*, pages 218–231. ACM, 2011.

[14] B. Billet and V. Issarny. Dioptase: a distributed data streaming middleware for the future web of things. *Journal of Internet Services and Applications*, 5:13, 2014.

[15] D. Booth, H. Haas, F. McCabe, E. Newcomer, M. Champion, C. Ferris, and D. Orchard. Web services architecture. 2004.

[16] C. Bormann, M. Ersue, and A. Keranen. Terminology for constrained-node networks. RFC 7228, IETF, May 2014.

[17] M. Botts and A. Robin. OGC SensorML: Model and XML Encoding Standard. 2014.

[18] A. Boulis, C. C. Han, R. Shea, and M. B. Srivastava. SensorWare: Programming sensor networks beyond code update and querying. *Pervasive and Mobile Computing*, 3:386–412, 2007.

[19] N. Brouwers, K. Langendoen, and P. Corke. Darjeeling, a feature-rich VM for the resource poor. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '07)*, pages 169–182, Berkeley, CA, USA, 2009. ACM.

[20] S. Brown and C. Sreenan. Software Updating in Wireless Sensor Networks: A Survey and Lacunae. *Journal of Sensor and Actuator Networks*, 2:717–760, 2013.

[21] P. Buonadonna, J. Hill, and D. Culler. Active message communication for tiny networked sensors. In *Proceedings of the 20th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM'01)*, pages 1–11, 2001.

[22] E. Cañete, J. Chen, M. Díaz, L. Llopis, and B. Rubio. A service-oriented approach to facilitate WSAN application development. *Ad Hoc Networks*, 9(3):430–452, may 2011.

[23] H. C. H. Cha, S. C. S. Choi, I. J. I. Jung, H. K. H. Kim, H. S. H. Shin, J. Y. J. Yoo, and C. Y. C. Yoon. RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks. In *Proceedings of 6th International Symposium on Information Processing in Sensor Networks (IPSN '07)*, pages 148–157, 2007.

[24] M. Compton, P. Barnaghi, L. Bermudez, R. García-Castro, O. Corcho, S. Cox, J. Graybeal, M. Hauswirth, C. Henson, A. Herzog, V. Huang, K. Janowicz, W. D. Kelsey, D. Le Phuoc, L. Lefort, M. Leggieri, H. Neuhaus, A. Nikolov, K. Page, A. Passant, A. Sheth, and K. Taylor. The SSN ontology of the W3C semantic sensor network incubator group. *Journal of Web Semantics*, 17:25–32, 2012.

[25] P. Costa, G. Coulson, R. Gold, M. Lad, C. Mascolo, L. Mottola, G. P. Picco, T. Sivaharan, N. Weerasinghe, and S. Zachariadis. The RUNES middleware for networked embedded systems and its application in a disaster management scenario. In *Proceedings of the 5th Annual IEEE International Conference on Pervasive Computing and Communications (PerCom '07)*, pages 69–78, 2007.

[26] P. Costa, G. Coulson, C. Mascolo, G. P. Picco, and S. Zachariadis. The RUNES middleware: a reconfigurable component-based approach to networked embedded systems. In *IEEE 16th International Symposium on Personal, Indoor and Mobile Radio Communications (PIMRC '05)*, pages 806–810, 2005.

[27] G. Coulson, G. Blair, P. Grace, F. Taiani, A. Joolia, K. Lee, J. Ueyama, and T. Sivaharan. A generic component model for building systems software. *ACM Transactions on Computer Systems*, 26(1):1:1–1:42, 2008.

[28] G. Cugola and A. Margara. SLIM: Service Location and Invocation Middleware for Mobile Wireless Sensor and Actuator Networks. *International Journal of Systems and Service-Oriented Engineering*, 1(3):60–74, 2010.

[29] W. Daniels, P. J. del Cid Garcia, W. Joosen, and D. Hughes. Safe reparametrization of component-based WSNs. In *Proceedings of the 10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2013)*, pages 524–536, Tokyo, Japan, 2014. Springer International Publishing.

[30] L. M. S. De Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio. SOCRADES: A Web service based shop floor integration infrastructure. In *Proceedings of the 1st International Conference on Internet of Things (IOT '08)*, volume 4952 LNCS, pages 50–67, 2008.

[31] A. Dearle, D. Balasubramaniam, J. Lewis, and R. Morrison. A component-based model and language for wireless sensor network applications. In *Proceedings of the 32nd Annual IEEE International Conference on Computer Software and Applications (COMPSAC '08)*, pages 1303–1308. IEEE, 2008.

[32] A. Dearle and S. Dobson. Mission-oriented middleware for sensor-driven scientific systems. *Journal of Internet Services and Applications*, 3(1):133–139, dec 2011.

[33] P. J. del Cid. *Middleware for Shared and Always-on Embedded Devices Extending component models to ease development effort and improve resource efficiency.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, KU Leuven, nov 2014.

[34] P. J. del Cid, D. Hughes, S. Michiels, and W. Joosen. Ensuring Application Integrity in Shared Sensing Environments. In *Proceedings of the 17th International ACM Sigsoft Symposium on Component-based Software Engineering (CBSE '14)*, pages 149–158, Marcq-en-Bareul, France, 2014. ACM.

[35] P. J. del Cid, N. Matthys, D. Hughes, S. Michiels, and W. Joosen. ACS: Specifying Smart Applications Using Sense-Process-Consume Flows. In *IEEE 12th International Symposium on Network Computing and Applications (NCA '13)*, pages 251–254, 2013.

[36] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*, pages 15–28, 2006.

[37] A. Dunkels, R. Gold, S. A. Marti, A. Pears, and M. Uddenfeldt. Janus : An Architecture for Flexible Access to Sensor Networks. In *Proceedings of the 1st ACM workshop on Dynamic interconnection of networks*, pages 48–52. ACM, 2005.

[38] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. *29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, 2004.

[39] A. Dunkels, O. Schmidt, and T. Voigt. Using protothreads for sensor node programming. *In Proceedings of the Workshop on RealWorld Wireless Sensor Networks (REALWSN '05)*, 2005.

[40] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads : Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, pages 29–42, Boulder, Colorado, USA, 2006. ACM.

[41] T. Erl. *Service-Oriented Architecture - Concepts, Technology, and Design.* Prentice Hall PTR, 2005.

[42] T. Erl. *SOA Principles of Service Design (The Prentice Hall Service-Oriented Computing Series from Thomas Erl).* Prentice Hall PTR, Upper Saddle River, NJ, USA, 2007.

[43] P. T. Eugster. Type-based Publish/Subscribe: Concepts and Experiences. *ACM Transactions on Programming Languages and Systems*, 29(1), 2007.

[44] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys*, 35(2):114–131, jun 2003.

[45] L. Evers, M. Bijl, and M. Marin-Perianu. Wireless sensor networks and beyond: A case study on transport and logistics. In *In International Workshop on Wireless Ad-Hoc Networks (IWWAN '05)*, pages 1–6, 2005.

[46] M. S. Familiar, J. F. Martínez, I. Corredor, and C. García-Rubio. Building service-oriented Smart Infrastructures over Wireless Ad Hoc Sensor Networks: A middleware perspective. *Computer Networks*, 56(4):1303–1328, mar 2012.

[47] M. S. Familiar, J. F. Martínez, and L. López. Pervasive Smart Spaces and Environments: A Service-Oriented Middleware Architecture for Wireless Ad Hoc and Sensor Networks. *International Journal of Distributed Sensor Networks*, pages 1–11, 2012.

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

[49] N. Finne, J. Eriksson, N. Tsiftes, A. Dunkels, and T. Voigt. Improving Sensornet Performance by Separating System Configuration from System Logic. In *Proceedings of the 7th European Conference on Wireless Sensor Networks (EWSN '10)*, pages 194–209, Coimbra, Portugal, 2010. Springer-Verlag.

[50] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS'05)*, pages 653–662, Columbus, Ohio, 2005. IEEE.

[51] C.-L. Fok, G.-C. Roman, and C. Lu. Enhanced coordination in sensor networks through flexible service provisioning. In *Proceedings of the 11th International Conference on Coordination Models and Languages (COORDINATION '09)*, pages 66–85, Lisboa, Portugal, 2009. Springer-Verlag.

[52] C.-L. Fok, G.-C. Roman, and C. Lu. Adaptive service provisioning for enhanced energy efficiency and flexibility in wireless sensor networks. *Science of Computer Programming*, 78(2):195–217, feb 2013.

[53] C. Frank and H. Karl. Consistency challenges of service discovery in mobile ad hoc networks. *Proceedings of the 7th ACM international symposium on Modeling, analysis and simulation of wireless and mobile systems (MSWiM '04)*, page 105, 2004.

[54] C. Frank and K. Römer. Algorithms for generic role assignment in wireless sensor networks. *Proceedings of the 3rd international conference on Embedded networked sensor systems (SenSys '05)*, pages 230–242, 2005.

[55] Gartner, Inc. Gartner Says 4.9 Billion Connected "Things" Will Be in Use in 2015. http://www.gartner.com/newsroom/id/2905717.

[56] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation (PLDI '03)*, pages 1–11, San Diego, California, USA, 2003. ACM.

[57] Google. Protocol Buffers. `https://developers.google.com/protocol-buffers/`. (visited July 2015).

[58] P. Grace, D. Hughes, B. Porter, G. S. Blair, G. Coulson, and F. Taiani. Experiences with open overlays: a middleware approach to network heterogeneity. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (Eurosys '08)*, pages 123–136, 2008.

[59] M. Grand. *Java Enterprise Design Patterns: Patterns in Java.* John Wiley & Sons, 2002.

[60] S. Hachem, A. Pathak, and V. Issarny. Service-Oriented Middleware for the Mobile Internet of Things : A Scalable Solution. In *IEEE Global*

Communications Conference (GLOBECOM '14), Austin, Texas, USA, 2014.

[61] S. Hadim and N. Mohamed. Middleware: Middleware Challenges and Approaches for Wireless Sensor Networks. *IEEE Distributed Systems Online*, 7(3):1–1, mar 2006.

[62] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. A Dynamic Operating System for Sensor Nodes. In *International Conference on Mobile Systems, Applications, and Services (MobiSys '05)*, pages 163–176, Seattle, Washington, USA, 2005. ACM.

[63] C.-C. Han, R. Kumar, R. Shea, and M. Srivastava. Sensor network software update management: a survey. *International Journal of Network Management*, 15:283–294, 2005.

[64] G. H. Hardy and E. M. Wright. *An introduction to the theory of numbers.* Oxford University Press, 1979.

[65] K. Hartke. Observing resources in the constrained application protocol (coap). RFC 7641, IETF, September 2015.

[66] J. Hauer, V. Handziski, A. Köpke, A. Willig, and A. Wolisz. A component framework for content-based publish/subscribe in sensor networks. In *Proceedings of the 5th European conference on Wireless Sensor Networks (EWSN '08)*, pages 369–385. Springer-Verlag, 2008.

[67] J. L. Hill and D. E. Culler. Mica: a wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, nov 2002.

[68] W. Horré. *Management Solutions for Distributed Software Applications in Multi-Purpose Sensor Networks (Beheersoplossingen voor gedistribueerde softwaretoepassingen in sensornetwerken voor meerdere doeleinden).* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, KU Leuven, oct 2011.

[69] W. Horré, S. Michiels, W. Joosen, and D. Hughes. Advanced sensor network software deployment using application-level quality goals. *Journal of Software*, 6(4):528–535, 2011.

[70] D. Hughes, P. Greenwood, G. Blair, G. Coulson, P. Grace, F. Pappenberger, P. Smith, and K. Beven. An experiment with reflective middleware to support grid-based flood monitoring. *Concurrency and Computation: Practice and Experience*, 20(11):1303–1316, aug 2008.

[71] D. Hughes, K. Thoelen, and W. Horré. LooCI: a loosely-coupled component infrastructure for networked embedded systems. In *Proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia (MoMM '09)*, pages 195–203, Kuala Lumpur, Malaysia, 2009. ACM.

[72] D. Hughes, K. Thoelen, W. Horré, N. Matthys, J. D. Cid, S. Michiels, C. Huygens, W. Joosen, and J. Ueyama. Building Wireless Wireless Sensor Network Applications with LooCI. *International Journal of Mobile Computing and Multimedia Communications*, 2(4):38–64, 2010.

[73] D. Hughes, K. Thoelen, J. Maerien, N. Matthys, W. Horre, P. J. del Cid Garcia, C. Huygens, S. Michiels, and W. Joosen. LooCI: The Loosely-coupled Component Infrastructure. In *11th IEEE International Symposium on Network Computing and Applications (NCA '12)*, pages 236–243. IEEE, aug 2012.

[74] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the 2nd international conference on Embedded networked sensor systems (SenSys '04)*, pages 81–94, Baltimore, Maryland, USA, 2004. ACM Press.

[75] iMinds AdMid project. Adaptive Middleware for Logistics. `http://www.iminds.be/en/projects/2014/03/05/admid`. (visited July 2015).

[76] iMinds COMACOD project. Control and management of constrained devices. `https://distrinet.cs.kuleuven.be/research/projects/COMACOD`. (visited July 2015).

[77] iMinds-DistriNet KU Leuven. LooCI webpages. `https://distrinet.cs.kuleuven.be/software/looci/`. (visited July 2015).

[78] iMinds MultiTr@ns project. Multimodal transport - mobility and logistics. `http://www.iminds.be/en/projects/2014/03/18/multitrans`. (visited July 2015).

[79] IPSO Alliance. IPSO Smart Object Guideline: Smart Objects Starter Pack 1.0. `http://challenge.ipso-alliance.org/so-starter-pack`. (visited November 2015).

[80] IWT STADiUM project. Software Technology for Adaptable Distributed Middleware. `https://distrinet.cs.kuleuven.be/projects/stadium/`. (visited July 2015).

[81] Java protocol buffers nano runtime library. `https://github.com/google/protobuf/tree/master/javanano`. (visited August 2015).

[82] C. Jennings, Z. Shelby, J. Arkko, and A. Keranen. Media types for sensor markup language (senml). draft, IETF, October 2015.

[83] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *First Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks (SECON '04)*, 2004.

[84] J. Jeong, S. Kim, and A. Broad. Network reprogramming. `http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/NetworkReprogramming.pdf`. (visited November 2015).

[85] P. Juang, H. Oki, Y. Wang, and M. Martonosi. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experiences with ZebraNet. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems*, San Jose, CA, 2002. ACM.

[86] J. Kahn, R. Katz, and K. Pister. Next century challenges: mobile networking for "Smart Dust". In *Proceedings of the 5th Annual ACM/IEEE International Conference on Mobile Computing and Networking*, pages 271–278, Seattle, Washington, USA, 1999. ACM.

[87] M. Karr and D. B. Loveman. Incorporation of units into programming languages. *Communications of the ACM*, 21(5):385–391, 1978.

[88] A. Kennedy. Dimension types. In *Proceedings of the 5th European Symposium on Programming: Programming Languages and Systems (ESOP '94)*, pages 348 –362. Springer-Verlag, 1994.

[89] J. King, R. Bose, S. Pickles, A. Helal, S. V. Ploeg, and J. Russo. Atlas: A Service-Oriented Sensor Platform Hardware and Middleware to Enable Programmable Pervasive Spaces. In *Proceedings of the 31st IEEE Conference on Local Computer Networks*, pages 630–638, 2006.

[90] K. Klues, C.-J. M. Liang, J. Paek, R. Musaloiu-E, P. Levis, A. Terzis, and R. Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys '09)*, pages 127–140, Berkeley, CA, USA, 2009. ACM.

[91] R. Koodli and C. E. Perkins. Service discovery in on-demand ad hoc networks. Internet-draft, IETF, October 2002.

[92] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, volume 2005, pages 354–365. IEEE, 2005.

[93] A. Kovacevic, J. Ansari, and P. Mahonen. NanoSD: A Flexible Service Discovery Protocol for Dynamic and Heterogeneous Wireless Sensor Networks. In *Sixth International Conference on Mobile Ad-hoc and Sensor Networks (MSN '10)*, pages 14–19. IEEE, 2010.

[94] D. Krafzig, K. Banke, and D. Slama. *Enterprise SOA: service-oriented architecture best practices.* Prentice Hall Professional, 2005.

[95] J. Kramer and J. Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, apr 1985.

[96] S. Kulkarni and L. W. L. Wang. MNP: Multihop Network Reprogramming Service for Sensor Networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems (ICDCS '05)*, pages 7–16. IEEE, 2005.

[97] M. Kushwaha, I. Amundson, X. Koutsoukos, S. Neema, and J. Sztipanovits. OASiS: A Programming Framework for Service-Oriented Sensor Networks. In *2nd International Conference on Communication Systems Software and Middleware (COMSWARE '07)*, pages 1–8. IEEE, jan 2007.

[98] K. Langendoen, A. Baggio, and O. Visser. Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th IEEE International Parallel & Distributed Processing Symposium (IPDPS '06)*. IEEE, 2006.

[99] B. Le Corre, J. Leguay, M. Lopez-Ramos, V. Gay, and V. Conan. Service Oriented Tasking System for WSN. In *Developments in E-systems Engineering (DESE '10)*, pages 64–69. IEEE, sep 2010.

[100] T. Lee, J. Hendler, and O. Lassila. The Semantic Web. *Scientific American*, 284(5):28–37, 2001.

[101] J. Leguay, M. Lopez-Ramos, K. Jean-Marie, and V. Conan. An efficient service oriented architecture for heterogeneous and dynamic wireless sensor networks. In *33rd IEEE Conference on Local Computer Networks (LCN '08)*, pages 740–747. IEEE, oct 2008.

[102] I. Leontiadis, C. Efstratiou, and C. Mascolo. SenShare: Transforming Sensor Networks into Multi-application Sensing Infrastructures. In *Proceedings of the 9th European Conference on Wireless Sensor Networks (EWSN '12)*, pages 65–81, Trento, Italy, 2012. Springer-Verlag.

[103] P. Levis. Experiences from a Decade of Development. In *Proceedings of the 10th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*, pages 207–220. USENIX, 2012.

[104] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, pages 85–95. ACM, 2002.

[105] P. Levis, D. Gay, and D. Culler. Active Sensor Networks. In *Proceedings of the 2nd Conference on Symposium on Networked Systems Design & Implementation (NSDI '05)*, pages 343–356. USENIX Association, 2005.

[106] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and Others. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–148. Springer Berlin Heidelberg, 2005.

[107] P. Levis, N. Patel, D. Culler, and S. 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 '04)*, pages 15–28, San Franciso, CA, USA, 2004.

[108] S. Li, Y. Lin, S. Son, J. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. In *Proceedings of the 2nd International Conference on Information Processing in Sensor Networks (IPSN '03)*, pages 502–517, Palo Alto, CA, USA, 2003. Springer-Verlag.

[109] J. Liu and F. Zhao. Towards semantic services for sensor-rich information systems. In *2nd International Conference on Broadband Networks (BROADNETS '05)*, pages 44–51, Boston, Massachusetts, USA, 2005. IEEE.

[110] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(212):122–173, 2005.

[111] J. Maerien. *A Security Framework for Shared Networked Embedded Systems.* PhD thesis, Informatics Section, Department of Computer Science, Faculty of Engineering Science, 2015.

[112] P. Mainwaring and J. Polastre. Wireless Sensor Networks for Habitat Monitoring. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications (WSNA '02)*, pages 88–97, New York, New York, USA, 2002. ACM Press.

[113] P. J. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A flexible and efficient code update

mechanism for sensor networks. In *Proceedings of the Third European Conference on Wireless Sensor Networks (EWSN '06)*, pages 212–227, Zurich, Switzerland, 2006. Springer-Verlag.

[114] P. J. Marron, A. Lachenmann, D. Minder, J. Hähner, R. Sauter, and K. Rothermel. TinyCubus: A flexible and adaptive framework for sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, pages 278–289. IEEE, 2005.

[115] N. Matthys, C. Huygens, D. Hughes, J. Ueyama, S. Michiels, and W. Joosen. Policy-driven tailoring of sensor networks. In *Proceedings of the Sensor Systems and Software conference 2010 (S-CUBE '10)*, volume 57, pages 20–35. Springer Berlin Heidelberg, 2010.

[116] W. P. McCartney and N. Sridhar. Abstractions for Safe Concurrent Programming in Networked Embedded Systems. In *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems (SenSys '06)*, SenSys '06, pages 167–180, New York, NY, USA, 2006. ACM.

[117] P. K. McKinley, S. M. Sadjadi, E. P. Kasten, and B. H. C. Cheng. Composing adaptive software. *IEEE Computer*, 37:56–64, 2004.

[118] W. Merrill. Where is the return on investment in wireless sensor networks? *IEEE Wireless Communications*, 17(1):4–6, 2010.

[119] E. Meshkova, J. Riihij, F. Oldewurtel, C. Jardak, and P. M. Service-Oriented Design Methodology for Wireless Sensor Networks: A View through Case Studies. In *IEEE International Conference on Sensor Networks, Ubiquitous, and Trustworthy Computing (SUTC '08)*, pages 146–153. IEEE, 2008.

[120] S. Michiels, W. Horré, W. Joosen, and P. Verbaeten. DAViM: a dynamically adaptable virtual machine for sensor networks. In *Proceedings of the international workshop on Middleware for sensor networks (MidSens '06)*, pages 7–12, Melbourne, Australia, 2006. ACM.

[121] mig - message interface generator for nesC. `http://www.tinyos.net/tinyos-1.x/doc/nesc/mig.html`. (visited August 2015).

[122] N. Mohamed and J. Al-Jaroodi. A survey on service-oriented middleware for wireless sensor networks. *Service Oriented Computing and Applications*, 5(2):71–85, apr 2011.

[123] G. Montenegro, N. Kushalnagar, J. Hui, and D. Culler. Transmission of ipv6 packets over ieee 802.15.4 networks. RFC 4944, IETF, September 2007.

[124] L. Mottola and G. P. Picco. Logical neighborhoods: A programming abstraction for wireless sensor networks. In *2nd IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '06)*, volume 4026 LNCS, pages 150–168, San Franciso, California, USA, 2006. Springer Berlin Heidelberg.

[125] L. Mottola and G. P. Picco. Middleware for wireless sensor networks: an outlook. *Journal of Internet Services and Applications*, 3(1):31–39, nov 2011.

[126] L. Mottola and G. P. Picco. Programming wireless sensor networks. *ACM Computing Surveys*, 43(3):1–51, apr 2011.

[127] L. Mottola, G. P. Picco, and A. Amjad Sheikh. FiGaRo: Fine-grained software reconfiguration for wireless sensor networks. In *Lecture Notes in Computer Science - Proceedings of the 5th European Conference on Wireless Sensor Networks (EWSN '08)*, volume 4913 LNCS, pages 286–304, Bologna, Italy, 2008. Springer Berlin Heidelberg.

[128] Nanopb. `http://koti.kapsi.fi/jpa/nanopb/`. (visited August 2015).

[129] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming Ad-hoc Networks of Mobile and Resource-constrained Devices. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, pages 249–260. ACM, 2005.

[130] OASIS. Devices Profile for Web Services (DPWS) Version 1.1. `http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01`. (visited November 2015).

[131] OASIS. Web services security: Soap message security 1.1 (ws-security 2004). `https://www.oasis-open.org/committees/download.php/16790/wss-v1.1-spec-os-SOAPMessageSecurity.pdf`. (visited November 2015).

[132] OASIS. Web services topics (ws-topics). `https://docs.oasis-open.org/wsn/wsn-ws_topics-1.3-spec-os.pdf`. (visited August 2015).

[133] Object Management Group (OMG). Common object request broker architecture (corba) specification, version 3.3. `http://www.omg.org/spec/CORBA/3.3/Interfaces/PDF/`, November 2012. (visited November 2015).

[134] Open Mobile Alliance (OMA). Lightweight machine to machine technical specification (candidate version 1.0), November 2014.

[135] Oracle. Project Sun SPOT. `http://www.sunspotdev.org/`. (visited July 2015).

[136] Oracle. Sensor edge server guide. http://otndnld.oracle.co.jp/document/products/as10g/101300/B25221_03/wireless.1013/b25142/title.htm. (visited November 2015).

[137] OSGi Alliance. The Dynamic Module System for Java. http://www.osgi.org. (visited July 2015).

[138] M. P. Papazoglou. Service-Oriented Computing : Concepts, Characteristics and Directions. In *Proceedings of the Fourth International Conference on Web Information Systems Engineering (WISE '03)*, pages 3–12. IEEE, 2003.

[139] M. P. Papazoglou and D. Georgakopoulos. Introduction: Service-oriented Computing. *Communications of the ACM ACM*, 46(10):24–28, oct 2003.

[140] M. P. Papazoglou, K. Pohl, M. Parkin, and A. Metzger, editors. *Service Research Challenges and Solutions for the Future Internet - S-Cube - Towards Engineering, Managing and Adapting Service-Based Systems*, volume 6500 of *Lecture Notes in Computer Science*. Springer, 2010.

[141] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

[142] C. E. Perkins, E. M. Belding-Royer, and S. R. Das. Ad hoc on-demand distance vector (aodv) routing. RFC 3561, IETF, July 2003.

[143] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling Ultra-low Power Wireless Research. In *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*, Los Angeles, California, 2005. IEEE.

[144] B. Porter, G. Coulson, and U. Roedig. Managing software evolution in large-scale wireless sensor and actuator networks. *ACM Transactions on Sensor Networks*, 9(4):1–28, 2013.

[145] B. Porter, U. Roedig, and G. Coulson. Type-safe updating for modular WSN software. In *International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS '11)*, pages 1–8. IEEE, 2011.

[146] D. Preuveneers and Y. Berbers. Prime numbers considered useful: ontology encoding for efficient subsumption testing. *Technical Report (CW Reports), Department of Computer Science, KU Leuven*, CW464, 2006.

[147] D. Preuveneers and Y. Berbers. Encoding semantic awareness in resource-constrained devices. *IEEE Intelligent Systems*, 23(2):26–33, 2008.

[148] N. Priyantha, A. Kansal, M. Goraczko, and F. Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems (SenSys '08)*, pages 253–266, Raleigh, North Carolina, USA, 2008. ACM.

[149] Project Haystack. `http://project-haystack.org/`. (visited August 2015).

[150] G. S. Ramachandran, W. Daniels, N. Matthys, C. Huygens, S. Michiels, W. Joosen, J. Meneghello, K. Lee, E. Canete, M. D. Rodriguez, and D. Hughes. Measuring and Modeling the Energy Cost of Reconfiguration in Sensor Networks. *IEEE Sensors Journal*, 15(6):3381–3389, 2015.

[151] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless Sensor Networks and Applications (WSNA '03)*, pages 60–67. ACM, 2003.

[152] A. Rezgui and M. Eltoweissy. Service-oriented sensor–actuator networks: Promises, challenges, and the road ahead. *Computer Communications*, 30(13):2627–2648, sep 2007.

[153] S. Rooney, D. Bauer, and P. Scotton. Edge server software architecture for sensor applications. In *Proceedings of the 2005 Symposium on Applications and the Internet (SAINT '05)*, pages 64–71, 2005.

[154] Z. Shelby. Constrained restful environments (core) link format. RFC 6690, IETF, August 2012.

[155] Z. Shelby and C. Chauvenet. The IPSO Application Framework. `http://www.ipso-alliance.org/wp-content/media/draft-ipso-app-framework-04.pdf`. (visited November 2015).

[156] Z. Shelby, K. Hartke, and C. Bormann. The constrained application protocol (coap). RFC 7252, IETF, June 2014.

[157] Z. Shelby, M. Koster, C. Bormann, and P. van der Stok. Core resource directory. Internet draft, IETF, October 2015.

[158] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh. Hourglass: An Infrastructure for Connecting Sensor Networks and Applications. Technical report, Harvard, 2004.

[159] S. K. Singh, M. P. Singh, and D. K. Singh. Routing protocols in wireless sensor networks. *International Journal of Computer Science & Engineering Survey (IJCSES)*, 1:63–83, 2010.

[160] SmartSantander. `http://www.smartsantander.eu/`. (visited October 2015).

[161] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal and Ubiquitous Computing*, 10(1):37–44, oct 2005.

[162] A. Stanford-Clark and H. Linh Truong. Mqtt for sensor networks (mqtt-sn) protocol specification v1.2. `http://mqtt.org/new/wp-content/uploads/2009/06/MQTT-SN_spec_v1.2.pdf`. (visited November 2015).

[163] J. Steffan and L. Fiege. Towards multi-purpose wireless sensor networks. In *Proceedings of the 2005 Systems Communications*, pages 336–341. IEEE, 2005.

[164] J. Z. Sun. Dissemination protocols for reprogramming wireless sensor networks: A literature survey. In *Proceedings of the 4th International Conference on Sensor Technologies and Applications (SENSORCOMM '10)*, pages 151–156, Venice, 2010. IEEE.

[165] C. Szyperski. *Component Software - Beyond Object-Oriented Programming.* Addison-Wesley/ACM Press, 2nd edition, 2002.

[166] A. Taherkordi, F. Loiret, A. Abdolrazaghi, R. Rouvoy, Q. Le-Trung, and F. Eliassen. Programming Sensor Networks Using REMORA Component Model. In *Proceedings of the 6th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '10)*, pages 45–62, Santa Barbara, CA, USA, 2010. Springer-Verlag.

[167] A. Taherkordi, F. Loiret, R. Rouvoy, and F. Eliassen. Optimizing Sensor Network Reprogramming via in-situ reconfigurable components. *ACM Transactions on Sensor Networks (TOSN)*, 9(2):1–37, 2013.

[168] K. Thoelen, D. Hughes, N. Matthys, L. Fang, S. Dobson, Y. Qiang, W. Bai, K. L. Man, S. U. Guan, D. Preuveneers, S. Michiels, C. Huygens, and W. Joosen. A reconfigurable component model with semantic type system for dynamic WSN applications. *Journal of Internet Services and Applications*, 3(3):277–290, 2012.

[169] K. Thoelen, D. Hughes, S. Michiels, and W. Joosen. StIgMa: Status information management for evolvable wireless sensor networks. In *Proceedings of the IEEE 3rd International Conference on Networked Embedded Systems for Every Application (NESEA '12)*, pages 1–7, Liverpool, UK, dec 2012. IEEE.

[170] K. Thoelen, S. Michiels, and W. Joosen. On-demand attribute-based service discovery for mobile WSANs. In *Proceedings of the 5th International Conference on Communication System Software and Middleware (COMSWARE '11)*, pages 1–6, Verona, Italy, 2011. ACM Press.

[171] K. Thoelen, D. Preuveneers, S. Michiels, W. Joosen, and D. Hughes. Types in their prime: sub-typing of data in resource constrained environments. In *Proceedings of the 10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2013)*, pages 250–261, Tokyo, Japan, 2013. Springer International Publishing.

[172] C. Timm, J. Schmutzler, P. Marwedel, and C. Wietfeld. Dynamic web service orchestration applied to the device profile for web services in hierarchical networks. In *Proceedings of the Fourth International ICST Conference on COMmunication System softWAre and middlewaRE (COMSWARE '09)*, pages 1–6, Dublin, Ireland, 2009. ACM.

[173] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, pages 121–132, 2005.

[174] P. Von Rickenbach and R. Wattenhofer. Decoding code on a sensor node. In *Proceedings of the 4th IEEE International Conference on Distributed Computing in Sensor Systems (DCOSS '08)*, volume 5067 LNCS, pages 400–414. Springer-Verlag, 2008.

[175] M. Wand and P. O'Keefe. Automatic Dimensional Inference. In *Computational Logic - Essays in Honor of Alan Robinson*, pages 479–483. 1991.

[176] M. Wang, J. Cao, J. Li, and S. Dasi. Middleware for wireless sensor networks: A survey. *Journal of computer science and Technology*, 23(2006):305–326, 2008.

[177] Q. Wang, Y. Zhu, and L. Cheng. Reprogramming wireless sensor networks: challenges and approaches. *IEEE Network*, 20(June):48–55, 2006.

[178] T. Watteyne, A. Molinaro, M. G. Richichi, and M. Dohler. From MANET to IETF ROLL standardization: A paradigm shift in WSN routing protocols. *IEEE Communications Surveys and Tutorials*, 13(4):688–707, 2011.

[179] M. Welsh and G. Mainland. Programming Sensor Networks Using Abstract Regions. In *Proceedings of the 1st Conference on Symposium on Networked Systems Design and Implementation - Volume 1 (NSDI '04)*, San Franciso, California, USA, 2004. USENIX Association.

[180] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *Proceeedings of the Second European Workshop on Wireless Sensor Networks (EWSN '05)*, pages 108–120. IEEE, 2005.

[181] G. Werner-Allen, K. Lorincz, M. Welsh, J. Johnson, and J. Lees. Deploying a Wireless Sensor Network on an Active Volcano. *IEEE Internet Computing*, 10(April):18–25, 2006.

[182] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services (MobiSys '04)*, pages 99–110, Boston, Massachusetts, USA, 2004. ACM.

[183] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *Proceedings of the 5th International Conference on Information Processing in Sensor Networks (IPSN '06)*, pages 416–423. IEEE, 2006.

[184] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J. Vasseur, and R. Alexander. Rpl: Ipv6 routing protocol for low-power and lossy networks. RFC 6550, IETF, March 2012.

[185] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(3):9, 2002.

[186] M. D. Yarvis, W. S. Conner, L. Krishnamurthy, J. Chhabra, B. Elliott, and A. Mainwaring. Real-World Experiences with an Interactive Ad Hoc Sensor Network. In *Proceedings of the Parallell Processing Workshops*, pages 143–151. IEEE, 2002.

[187] J. Ye, S. Dasiopoulou, G. Stevenson, G. Meditskos, E. Kontopoulos, I. Kompatsiaris, and S. Dobson. Semantic web technologies in pervasive computing: A survey and research roadmap. *Pervasive and Mobile Computing*, 23:543–549, 2015.

[188] Y. Yu, B. Krishnamachari, and V. Prasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18(1):15–21, 2004.

[189] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *Proceedings of the 4th international conference on Embedded networked sensor systems (SenSys '06)*, pages 139–152, Boulder, Colorado, USA, 2006. ACM Press.

[190] E. Zeeb, G. Moritz, D. Timmermann, and F. Golatowski. WS4D: Toolkits for Networked Embedded Systems Based on the Devices Profile for Web Services. In *39th International Conference on Parallel Processing - Workshops (ICPPW '10)*, pages 1–8, San Diego, California, USA, 2010. IEEE.

[191] ZigBee Alliance. ZigBee Specification. `http://www.zigbee.org/`. (visited July 2015).

List of publications

Articles in international peer reviewed journals

Klaas Thoelen, Danny Hughes, Nelson Matthys, Lei Fang, Simon Dobson, Yizhou Qiang, Wei Bai, Ka Lok Man, Sheng-Uei Guan, Davy Preuveneers, Sam Michiels, Christophe Huygens, Wouter Joosen. A reconfigurable component model with semantic type system for dynamic wsn applications. *Journal of Internet Services and Applications*, volume 3, issue 3, pages 277-290, 01 December 2012

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

Contributions at international conferences, published in proceedings

Klaas Thoelen, Wouter Joosen, Danny Hughes. Putting sense inside sensor systems: a coordinated approach to messaging. *Proceedings of the 14th IEEE International Symposium on Network Computing And Applications (NCA 2015)*, pages 22-26, Cambridge, MA, USA, 28-30 September 2015

Klaas Thoelen, Davy Preuveneers, Sam Michiels, Wouter Joosen, Danny Hughes. Types in their prime: sub-typing of data in resource constrained environments. *Proceedings of the 10th International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (Mobiquitous 2013)*, pages 250-261, Tokyo, Japan, 2-4 December 2013

Klaas Thoelen, Danny Hughes, Sam Michiels, Wouter Joosen. StIgMa: Status information management for evolvable wireless sensor networks. *Proceedings of the 3rd IEEE International Conference on Networked Embedded Systems for Every Application (NESEA 2012)*, pages 1-7, Liverpool, UK, 13-14 December 2012

Danny Hughes, Klaas Thoelen, Jef Maerien, Nelson Matthys, Pedro Javier del Cid Garcia, Wouter Horré, Christophe Huygens, Sam Michiels, Wouter Joosen. LooCI: the Loosely-coupled Component Infrastructure. *Proceedings of the 11th IEEE International Symposium on Network Computing And Applications (NCA 2012)*, pages 236-243, Cambridge, MA, USA, 23-25 August 2012

Klaas Thoelen, Sam Michiels, Wouter Joosen. On-demand attribute-based service discovery for mobile WSANs. *Proceedings of the 5th International Conference on Communication System Software and Middleware (COMSWARE 2011)*, Verona, Italy, 4-5 July 2011

Klaas Thoelen, Sam Michiels, Wouter Joosen. Middleware for adaptive group communication in wireless sensor networks. *Proceedings of the 2nd International ICST conference on Sensor Systems and Software (S-CUBE 2010)*, pages 59-74, Miami, Florida, USA, 13-14 December 2010

Klaas Thoelen, Nelson Matthys, Wouter Horré, Christophe Huygens, Wouter Joosen, Danny Hughes, Lei Fang, Sheng-Uei Guan. Supporting reconfiguration and re-use through self-describing component interfaces. *Proceedings of the 5th International Workshop on Middleware Tools, Services and Run-Time Support for Sensor Networks (MidSens 2010)*, pages 29-34, Bangalore, India, 30 November 2010

Danny 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. *Proceedings of the 7th International Conference on Advances in Mobile Computing & Multimedia (MoMM 2009)*, pages 195-203, Kualu Lumpur, Malaysia, 14-16 December 2009

Klaas Thoelen, Sam Michiels, Wouter Joosen, Koen Vangheluwe, Katja Verbeeck. A sensor middleware and agent-based communication platform for supply-chain management. *Proceedings of 8th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2009)*, pages 1409-1410, Budapest, Hungary, 10-15 May 2009

Klaas Thoelen, Sam Michiels, Wouter Joosen. Tracking and tracing containers through distributed sensor middleware. *Proceedings of the 2nd International Conference on Autonomic Computing and Communication Systems (Autonomics*

2008), pages 1-7, Turin, Italy, 23-25 September 2008

Klaas Thoelen, Sam Michiels, Wouter Joosen. A distributed middleware for container transport: lessons learned. *Proceedings of the 7th workshop on Middleware for Network Eccentric and Mobile Applications (MiNEMA 2008)*, pages 19-28, Lappeenranta, Finland, 21 August 2008

Technical reports

Danny 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, KU Leuven, September 2009

FACULTY OF ENGINEERING SCIENCE
DEPARTMENT OF COMPUTER SCIENCE
IMINDS - DISTRINET
Celestijnenlaan 200A box 2402
B-3001 Heverlee
https://distrinet.cs.kuleuven.be/