TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

Teemu Laukkarinen

Abstracting Application Development for Resource Constrained Wireless Sensor Networks

Julkaisu 1306 • Publication 1306

Tampere 2015

Tampereen teknillinen yliopisto. Julkaisu 1306
Tampere University of Technology. Publication 1306

Teemu Laukkarinen

Abstracting Application Development for Resource Constrained Wireless Sensor Networks

Thesis for the degree of Doctor of Science in Technology to be presented with due permission for public examination and criticism in Tietotalo Building, Auditorium TB104, at Tampere University of Technology, on the 4th of September 2015, at 12 noon.

Tampereen teknillinen yliopisto - Tampere University of Technology
Tampere 2015

ISBN 978-952-15-3542-0 (printed)
ISBN 978-952-15-3567-3 (PDF)
ISSN 1459-2045

ABSTRACT

Ubiquitous computing is a concept whereby computing is distributed across smart objects surrounding users, creating ambient intelligence. Ubiquitous applications use technologies such as the Internet, sensors, actuators, embedded computers, wireless communication, and new user interfaces. The Internet-of-Things (IoT) is one of the key concepts in the realization of ubiquitous computing, whereby smart objects communicate with each other and the Internet. Further, Wireless Sensor Networks (WSNs) are a sub-group of IoT technologies that consist of geographically distributed devices or nodes, capable of sensing and actuating the environment.

WSNs typically contain tens to thousands of nodes that organize and operate autonomously to perform application-dependent sensing and sensor data processing tasks. The projected applications require nodes to be small in physical size and low-cost, and have a long lifetime with limited energy resources, while performing complex computing and communications tasks. As a result, WSNs are complex distributed systems that are constrained by communications, computing and energy resources. WSN functionality is dynamic according to the environment and application requirements. Dynamic multitasking, task distribution, task injection, and software updates are required in field experiments for possibly thousands of nodes functioning in harsh environments.

The development of WSN application software requires the abstraction of computing, communication, data access, and heterogeneous sensor data sources to reduce the complexities. Abstractions enable the faster development of new applications with a better reuse of existing software, as applications are composed of high-level tasks that use the services provided by the devices to execute the application logic.

The main research question of this thesis is: *What abstractions are needed for application development for resource constrained WSNs*? This thesis models WSN abstractions with three levels that build on top of each other: 1) node abstraction, 2) network abstraction, and 3) infrastructure abstraction. The node abstraction hides the details in the use of the sensing, communication, and processing hardware. The network abstraction specifies methods of discovering and accessing services, and distributing processing in the network. The infrastructure abstraction unifies different sensing technologies and infrastructure computing platforms.

As a contribution, this thesis presents the abstraction model with a review of each abstraction level. Several designs for each of the levels are tested and verified with proofs of concept and analyses of field experiments. The resulting designs consist of an operating system kernel, a software update method, a data unification interface, and all abstraction levels combining abstraction called an embedded cloud.

The presented operating system kernel has a scalable overhead and provides a programming approach similar to a desktop computer operating system with threads and processes. An over-the-air update method combines low overhead and robust software updating with application task dissemination. The data unification interface homogenizes the access to the data of heterogeneous sensor networks. A unification model is used for various use cases by mapping everything as measurements. The embedded cloud allows resource constrained WSNs to share services and data, and expand resources with other technologies. The embedded cloud allows the distributed processing of applications according to the available services. The applications are implemented as processes using a hardware independent description language that can be executed on resource constrained WSNs. The lessons of practical field experimenting are analyzed to study the importance of the abstractions. Software complexities encountered in the field experiments highlight the need for suitable abstractions.

The results of this thesis are tested using proof of concept implementations on real WSN hardware which is constrained by computing power in the order of a few MIPS, memory sizes of a few kilobytes, and small sized batteries. The results will remain usable in the future, as the vast amount, tight integration, and low-cost of future IoT devices require the combination of complex computation with resource constrained platforms.

PREFACE

The research work for this thesis was carried out in the Department of Computer Systems and later in the Department of Pervasive Computing at Tampere University of Technology during the years 2009 – 2014.

I would like to express my sincere gratitude to my supervisor Adjunct Professor Marko Hännikäinen for his guidance, support, and motivation during the research. I am also grateful to Professor Timo D. Hämäläinen, Professor Tommi Mikkonen, and Professor Mikko Tiusanen for their help in the process. I would like to thank Professor Evgeny Osipov and Professor Jukka Riekki for reviewing my thesis. Also, I would like to thank Professor Ismo Hakala and Professor Jukka Riekki for agreeing to act as opponents in the defense.

Humongous thanks go to the members of the TUTWSN team for their valuable work that after all made this thesis possible. Special thanks to Dr. Mauri Kuorilehto, D.Sc., Dr. Mikko Kohvakka, D.Sc., Dr. Jukka Suhonen, D.Sc., Dr. Ville Kaseva, D.Sc., Mr. Timo Alho, M.Sc., Mr. Lasse Määttä, M.Sc., Mr. Olli Kivelä, M.Sc., Mr. Juha Onkila, M.Sc., Mr. Ilkka Kautto, M.Sc., Mr. Markku Hänninen, M.Sc., and the rest of colleagues in the departments that I have had pleasure to work with.

My thesis was financially supported by HPY Research Foundation, Tekniikan edistämissäätiö, Ulla Tuominen Foundation, and Walter Ahlström Foundation from which I am truly grateful.

I would like to express special gratitude to my family for their support and encouragement: my parents, Tuula and Martti, and my siblings, Tanja and Simo, and their families.

Also, I would like to thank all those persons, groups, and things that have kept me going while working on this thesis: my vital friends at #eetr and #evokit IRC channels; rest of my friends; Sakari Karipuro and rest of #kameramafia; L'Amusette and their companions; Alex Regan and Klubi; numerous musical acts and bands; several coffee shops; and some other functional interfaces. No animals were harmed in making of this thesis, regardless of someone asking about feeding the nodes to cows.

Finally, I would like to end this to a very inspirational quote, which describes in an exquisite way the process of making a doctoral thesis, but also many other aspects of life:

“WAGRRRRWWGAHHHHWWWRRGGAWWWWWWRR!”

- Chewbacca in Star Wars: The Empire Strikes Back

Tampere, May 12, 2015

Teemu Laukkarinen

TABLE OF CONTENTS

LIST OF PUBLICATIONS

This Thesis consists of an introductory part and the following six publications. In the introductory part the publications are referred to as [P1], [P2], ..., [P6].

[P1] T. Laukkarinen, V. A. Kaseva, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "HybridKernel: Preemptive Kernel with Event-driven Extension for Resource Constrained Wireless Sensor Networks", *IEEE Workshop on Signal Processing Systems*, October 7–9, 2009, Tampere, Finland. doi:10.1109/SIPS.2009.5336243

[P2] T. Laukkarinen, L. Määttä, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "Design and Implementation of a Firmware Update Protocol for Resource Constrained Wireless Sensor Networks". *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2(3), pp. 50–68, 2011. doi:10.4018/978–1–4666–2776–5.ch003

[P3] T. Laukkarinen, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "Pilot studies of wireless sensor networks: Practical experiences," *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–8, November 2–4, 2011, Tampere, Finland. doi:10.1109/DASIP.2011.6136867

[P4] T. Laukkarinen, J. Suhonen, and M. Hännikäinen, “A Survey of Wireless Sensor Network Abstraction for Application Development,” *International Journal of Distributed Sensor Networks*, vol. 2012, Article ID 740268, 12 pages, 2012. doi:10.1155/2012/740268

[P5] J. Suhonen, O. Kivela, T. Laukkarinen, and M. Hännikäinen, "Unified service access for wireless sensor networks," *Third International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pp. 49–55, June 2, 2012. doi:10.1109/SESENA.2012.6225735

[P6] T. Laukkarinen, J. Suhonen, and M. Hännikäinen, "An embedded cloud design for Internet-of-Things," *International Journal of Distributed Sensor Networks*, vol. 2013, Article ID 790130, 13 pages, 2013. doi:10.1155/2013/790130

LIST OF ABBREVIATIONS

6LoWPAN	IPv6 over Low power Wireless Personal Area Networks
ACF	Authentication and Capability Format
API	Application Programming Interface
CoAP	Constrained Application Protocol
CPU	Central Processing Unit
CSV	Comma Separated Values
DiMiWa	Distributed Middleware
EEPROM	Electronically Erasable Programmable Read-Only Memory
FMI	Finnish Meteorological Institute
GPRS	General Packet Radio Service
GSN	Global Sensor Networks
HTTP	Hypertext Transfer Protocol
HVAC	Heating, Ventilation, and Air Conditioning
HW	Hardware
IC	Integrated Circuit
IETF	Internet Engineering Task Force
IPC	Inter-Process Communication
IPv6	Internet Protocol version 6
I/O	Input/Output

IoT	Internet-of-Things
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
MAC	Medium Access Control
MEDF	Meta-Data Format
MEMS	Micro Electro Mechanical Systems
MCU	Micro Controller Unit
NaaS	Network as a Service
NASC	Node Actuator and Sensor Control
NMF	Network Management Format
O&M	Observations and Measurements
OGC	Open Geospatial Consortium
OS	Operating System
OWL	Web Ontology Language
OTAP	Over The Air Programming
PC	Personal Computer
PCB	Printed Circuit Board
PDA	Personal Digital Assistant
PDL	Process Description Language
PID	Proportional-Integral-Derivative
PIDP	Program Image Distribution Protocol
QoS	Quality of Service
RAM	Random Access Memory
REST	Representational State Transfer

RF	Radio Frequency
RTOS	Real-time Operating System
SADF	Sensor Archive Data Format
SAS	Sensor Alert Service
SaaS	Software as a Service
SensorML	Sensor Model Language
SES	Sensor Event Service
SIDF	Sensor Information Data Format
SIR	Sensor Instance Registry
SOA	Service-oriented Architecture
SOAP	Simple Object Access Protocol
SOS	Sensor Observation Service
SPS	Sensor Planning Service
SQL	Structured Query Language
SOR	Sensor Observation Registry
SSN	Semantic Sensor Network
SWE	Sensor Web Enablement
TDMA	Time-Division Multiple Access
TML	Transducer Markup Language
TTL	Time-To-Live
TUTWSN	Tampere University of Technology Wireless Sensor Network
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
UVPN	Ubiquitous Virtual Private Network

VM	Virtual Machine
W3C	World Wide Web Consortium
WNS	Web Notification Service
WOAG	WSN OpenAPI Gateway
WSDL	Web Services Description Language
WSN	Wireless Sensor Network
WWW	World Wide Web
XML	eXtensible Markup Language

1. INTRODUCTION

Ubiquitous computing is a vision whereby human-computer interaction is obscured, and an ambient intelligence experience is created for the user [214]. The Internet-of-Things (IoT) is a key enabling concept, whereby ubiquitous applications are created with the help of things that communicate and interact in cooperation to execute intelligent tasks [18]. The tasks can be anything from simple automated lighting to interactive social experiences that fuse information from various sources. The term IoT is ambiguous and lacks a clear textbook definition [18]. In this thesis, IoT is understood as heterogeneous smart devices that form a communication network to exchange data peer-to-peer through a common interface and the Internet.

Wireless Sensor Networks (WSNs) are a specific subset of IoT technologies, whereby spatially distributed sensing devices form a wireless network to deliver measurement data [4, 117, 221]. WSN devices are programmable small computers running software that allows measurement, data collection, data processing, and communication. WSNs are often bi-directional and can control appliances through actuators. The sensory data and actuating capability of WSNs plays a key role in creating intelligent ubiquitous applications. The constant development of small energy-efficient Integrated Circuits (ICs) and communication technologies, and the emergence of small energy-efficient Micro Electro Mechanical Systems (MEMS) sensors have enabled these WSNs [4, 38, 117, 221].

1.1 WSN Design Characteristics

The WSN paradigm states that the wireless devices, or *nodes*, autonomously form a network, and thus can be easily deployed in the physical world [4, 38, 117]. Figure 1 presents the WSN reference architecture: the nodes measure phenomena, communicate wirelessly in a multi-hop fashion, and deliver data to the gateways, where the data can be forwarded to the end users.

Many WSN deployments have a harsh environment that causes a physical burden

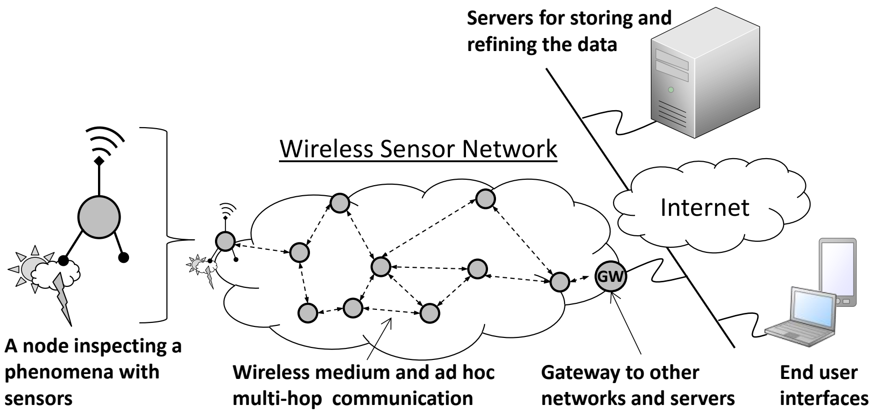

Servers for storing and refining the data
Wireless Sensor Network
Internet
GW
A node inspecting a phenomena with sensors
Wireless medium and ad hoc multi-hop communication
Gateway to other networks and servers
End user interfaces

Fig. 1: *The WSN reference architecture.*

on the nodes [4, 21, 38, 118, 190]. Deployments are located in mines, battlefields, greenhouses, ships, sewers or factories that have untypical conditions, e.g. extremely high or low humidity and temperature. The nodes may be exposed to high radiation in a nuclear disaster area or to extreme vibration when attached to machinery. Snow fall or falling leaves can change the Radio Frequency (RF) communication environment drastically. In static environmental conditions, other physical changes are evident: people and RF absorbing/reflecting objects move and vary in density, doors open and close, and vandalism is a potential issue. As a result, nodes appear, disappear, and reappear in WSN deployments and this must be considered in WSN application development.

The requirements of maintenance-free operation, small physical size, and low manufacturing costs are incorporated into WSNs, since networks of hundreds to thousands of nodes are expected to function for several years [4, 38, 117, 168]. Some visions suggest that the nodes could be disposable [38]. These requirements have resulted in *resource constrained WSNs*.

Resource constrained WSN nodes are constructed from a small Micro Controller Unit (MCU), a wireless communication device, sensors and actuators, and a limited energy source [4, 38, 117]. The communication, processing, and sampling must be optimized to achieve years of maintenance-free operation with batteries or energy harvesting devices, such as solar panels. The small MCUs have limited memory and processing resources, which makes development of the embedded WSN software challenging due to the incorporate complex protocols and distributed functionality. The research in this thesis concentrates on these resource constrained WSNs.

1.2 Research Questions

This thesis focuses on facilitating application development for resource constrained WSNs. Software abstractions are the chosen research approach, as these aim to make application development faster, simpler, less error prone, and less Hardware (HW) dependent [168]. These benefits are achieved when the abstraction reduces implementation complexity by hiding details from the application programmer [22, 106, 114, 168]. The abstraction separates applications with an interface, which allows portability and reuse [114, 152, 153].

For WSN application development, the resource constraints and the expectation of autonomous ad hoc networking in harsh conditions set up a complex system that requires WSN-specific abstractions [21, 118, 168]. Similarly, creating such abstractions is complicated due to the limited resources and distributed functionality.

The main research question of this thesis is: *What abstractions are needed for application development for resource constrained WSNs.* This question divides into more detailed questions as follows.

- How should one divide the abstractions hierarchically and what are the responsibilities of each level?
- How should one execute application tasks on an Operating System (OS) as neither the existing pre-emptive kernels nor event-driven kernels alone have the required characteristics?
- How should one disseminate new software and applications to the distributed WSN nodes?
- How should one homogenize the sensor data and actuator accessing for heterogeneous WSNs?
- How should one unify the functionality of a node, a network and an infrastructure for distributed processing over heterogeneous WSNs?

1.3 Scope and Contribution of Thesis

The scope and contribution of this thesis are presented in Figure 2. The scope is in the software abstractions and practical development of resource constrained WSN

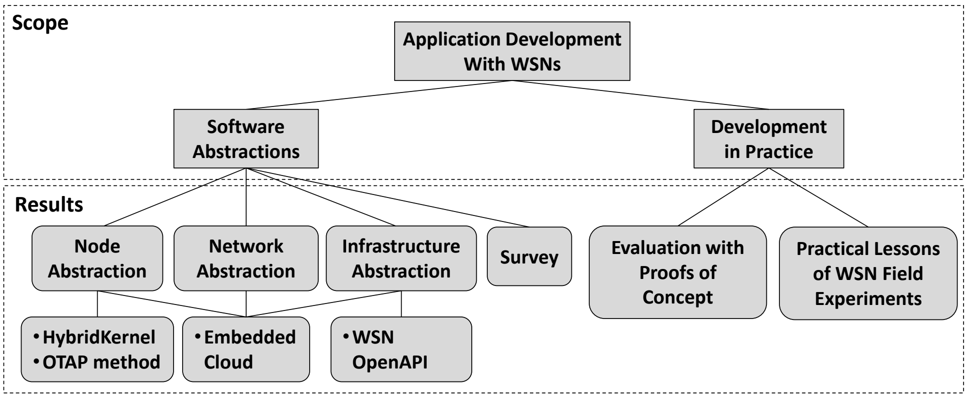


Fig. 2: *The Scope and the Contribution of this thesis.*

applications. The emphasis is on solving open issues set by the resource constraints, and testing the design feasibility in practice with real WSN HW.

This thesis models WSN application development with three abstraction levels as depicted in Figure 3: a) node abstraction, b) network abstraction, and c) infrastructure abstraction. The levels are explained in the following paragraphs.

The node abstraction hides HW specifics from WSN applications with an OS and a protocol stack. The OS provides an execution environment for the WSN applications. The protocol stack abstracts communication. WSN applications on the node abstraction include such tasks as reading a sensor, processing data, and sending data to an interested party. These tasks require energy-aware functioning to ensure the long lifetime of a battery-powered node. The node abstraction is implemented on the resource constrained nodes as embedded software. Thus, handling the constrained resources is the major design issue for the node abstraction designs.

The network abstraction hides the distributed nodes and provides such methods as service discovery, service access, and distributed processing. For example, a WSN application on a network abstraction can request an average temperature from a group of nodes or create an alarm at too high humidity. The network abstractions are implemented as software on the resource constrained nodes. However, parts of the software can be implemented on resource richer nodes, gateways, or servers. The network abstraction on the WSN nodes is typically called *middleware* [37,83,141,151]. Handling the constrained resources of the nodes and the distributed functioning in harsh environments are the major design issues for the network abstractions.

The infrastructure abstraction hides heterogeneous sensor networks and supporting technologies behind one unified interface. It is needed because one WSN technology

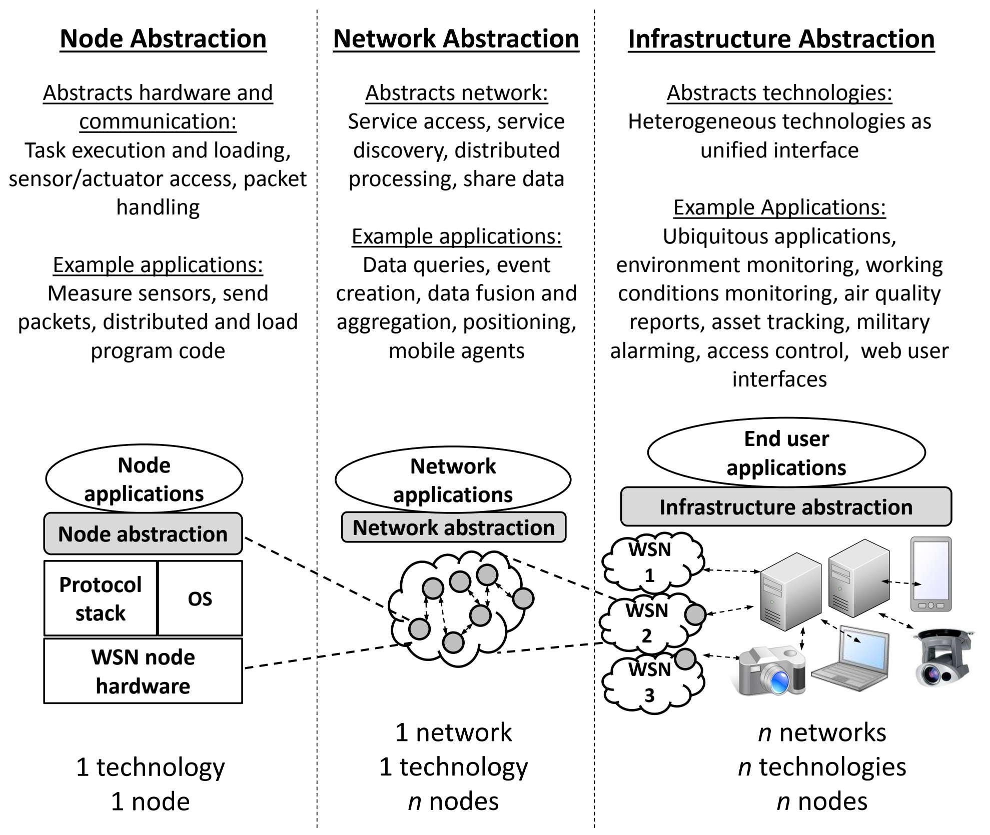

Node Abstraction
Abstracts hardware and communication:
Task execution and loading, sensor/actuator access, packet handling
Example applications:
Measure sensors, send packets, distributed and load program code
Network Abstraction
Abstracts network:
Service access, service discovery, distributed processing, share data
Example applications:
Data queries, event creation, data fusion and aggregation, positioning, mobile agents
Infrastructure Abstraction
Abstracts technologies:
Heterogeneous technologies as unified interface
Example Applications:
Ubiquitous applications, environment monitoring, working conditions monitoring, air quality reports, asset tracking, military alarming, access control, web user interfaces
Node applications
Node abstraction
Protocol stack
OS
WSN node hardware
Network applications
Network abstraction
End user applications
Infrastructure abstraction
WSN 1
WSN 2
WSN 3
1 technology
1 node
1 network
1 technology
n nodes
n networks
n technologies
n nodes

Fig. 3: *The WSN abstraction model of this thesis.*

is rarely capable of delivering all of the data that the end user application requires. For example, delivering multimedia data over a resource constrained WSN is not viable [5]. WSN measurements and actuators may be integrated with other existing systems, such as Heating, Ventilation, and Air Conditioning (HVAC) and lighting systems in a building. Hiding the WSN technology specific details makes the end user application development independent of the WSN technology and ensures portability. The infrastructure abstractions are implemented on resource rich computing platforms, such as embedded Personal Computers (PCs), servers, or cloud computing platforms. Handling the heterogeneous technologies, the vast application space, and the dynamic WSN services are the major design issues for the infrastructure abstractions.

The main contribution of this thesis is the abstraction model. In addition, the following contributions are made related to the abstraction model and given research questions.

- A survey of the software abstractions [P4] characterizes, classifies, and analyzes the on-going research on infrastructure abstraction.
- An embedded cloud design [P6] binds the three abstractions into one design, which allows resource, service, and processing distribution between heterogeneous sensor networks.
- HybridKernel [P1] allows scalable multitasking on resource constrained nodes.
- An Over The Air Programming (OTAP) method combines efficient firmware updating [P2] and application dissemination [P6].
- WSN OpenAPI [P5] is an eXtensible Markup Language (XML) specification for unified access to heterogeneous WSN measurements and actuators.
- Practical lessons of field experimenting are analyzed for WSN application development [P3].

1.4 Methods of Thesis

Three methods are used in this thesis: 1) literature survey, 2) proof of concept, and 3) analysis of field experiments. These methods are popular in information systems research [61].

A literature survey is used in [P4] to characterize the abstractions and to find open issues in WSN abstractions.

Proof of concept is used in [P1], [P2], [P5], and [P6]. In this thesis, proof of concept answers the question that "*Is it possible to implement the design on resource constrained WSN nodes?*" through functionality and feasibility metrics such as memory, energy, and execution overheads.

Field experiments are analyzed in [P2], [P3], and [P5]. A field experiment indicates that the design works in unpredictable environmental conditions and concludes lessons learned for the WSN abstractions and application development.

1.5 Thesis Outline

This thesis is constructed from an introductory part and six publications [P1 – P6]. The introductory part motivates, summarizes, and concludes the research work presented in this thesis. The publications present the main results of this thesis. The rest of the introductory part has three distinctive sections, organized as follows.

The first part consists of Chapter 2 and Chapter 3, which cover WSN application development with resource constrained WSN nodes. Chapter 2 gives the background of resource constrained nodes, node abstractions and network abstractions. Chapter 3 summarizes the thesis results for WSN application development in resource constrained nodes.

The second part consists of Chapters 4 and 5, which cover infrastructure abstractions for heterogeneous sensor networks. Chapter 4 presents a review of infrastructure abstractions. Chapter 5 summarizes the research results for infrastructure abstractions.

The third part is Chapter 6, which covers the analysis of lessons from field experiments with WSNs. The related research is given and the thesis results of the field experimenting summarized.

Chapter 7 summarizes the publications included in this thesis. Finally, Chapter 8 concludes the thesis and presents the discussion and future research work.

2. REVIEW ON NODE AND NETWORK ABSTRACTIONS FOR WSN NODES

This chapter summarizes knowledge of node and network abstractions for resource constrained WSN nodes. The related studies [37, 154, 179, 191] discuss WSN OS kernels designs, middleware, OTAP, and programming models as abstractions for the WSN applications on the nodes. These topics are mapped to the abstraction model in Figure 4. The protocol stacks are left out of the review as a separate research field.

2.1 Abstractions on WSN Nodes

Abstractions on the node are required for three reasons: 1) The WSN applications share the HW components in the node. Implementing an application is faster, if the HW access is the same for all of the WSN applications. 2) The node HW is heterogeneous as different types of nodes have different MCUs, sensors, actuators, and communication ICs. Without node abstraction, the same WSN application would have to be implemented multiple times for different HW combinations. 3) The node

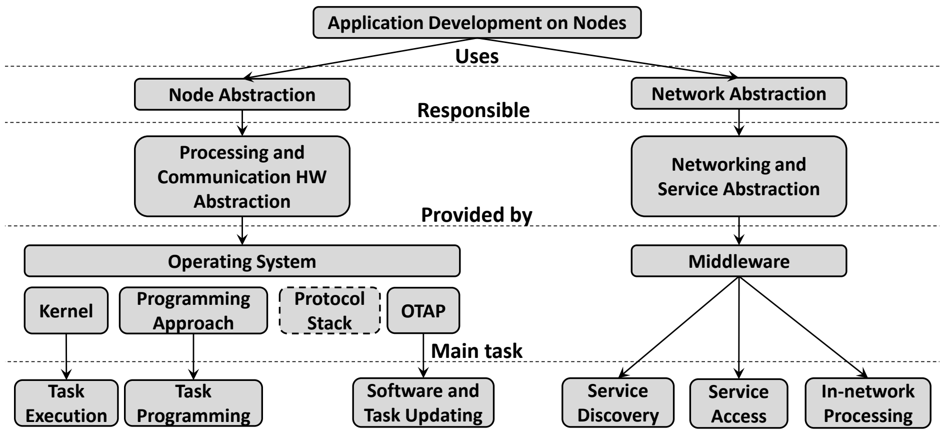


Fig. 4: *A mapping of the topics covered in this chapter.*

resources need management to achieve optimum operation and life-time. The abstraction should manage resource-sharing between the WSN applications.

The node abstraction consists of embedded software components that run on resource constrained WSN HW. Figure 5 presents a typical node HW platform and its software components. An OS kernel abstracts the Central Processing Unit (CPU), sensors, actuators, and communication HW specifics. A protocol stack hides the communication, acting as an interface to send and receive data between the nodes. OTAP allows software updates and application injection. Middleware abstracts the network over the distributed nodes and allows network-wide applications. These components are interconnected as one WSN OS that runs the WSN applications in the WSN nodes.

The software components that connect the larger units implement the abstraction as an interface. Therefore, this thesis concentrates on these interconnecting components: the OS kernel, OTAP, and middleware.

2.2 WSN Node Platforms

The WSN node platforms typically consist of an MCU, a radio with an antenna, a power supply, sensors, actuators, and a Printed Circuit Board (PCB) as depicted in Figure 5 [108]. The node platforms are restricted in computing, memory, and energy resources due to the size and expected life-time requirements. Table 1 presents currently available node platforms grouped by the MCUs used. The node platforms typically utilize two series connected AA batteries as an energy source. One 1.5 V AA -battery has a capacity of 1000 to 3000 mAh depending on the current draining and chemicals used [65]. This thesis uses 20000 J (ca. 2000 mAh at 3 V) as a reference energy budget.

2.2.1 Software Execution on MCUs

An MCU contains a CPU, program memory, data memory, Electronically Erasable Programmable Read-Only Memory (EEPROM), timers, and Input/Output (I/O) connections. The CPU executes embedded software from the integrated program memory. Therefore, MCUs set limitations on the abstraction designs that are implemented as embedded software on the nodes.

[1] The specification gives value of 5mA at 4 MHz. Given value is scaled assuming linear scaling.

Table 1: *Low power MCUs used in the existing resource constrained node platforms. Sleep currents are the minimum sleep state, where the MCU can wake up to continue execution without an external wake up signal.*

MCU	Program (kB)	Data (kB)	Active μA/MHz	Sleep μA	Platforms
Atmel ATmega328P [16]	32	2	300 @ 1.8 V	4.20	Arduino Uno [11]
Atmel ATmega2560 [15]	256	8	500 @ 1.8 V	<5.00	Arduino Mega [11]
Atmel ATmega128L [13]	128	4	1250 [1] @ 3 V	<15.00	MEMSIC MICAz [144] MICA2 [146] BTnode ver3 [227]
Atmel ATmega1281 [14]	128	8	500 @ 1.8 V	<5.00	Libelium Waspmote [129] MEMSIC IRIS [143]
Texas Instruments MSP430 [93]	48	10	330 @ 2.2 V	1.1	MEMSIC TelosB [145] Shimmer [172]
Microchip PIC18LF8722 [148]	128	3.9	380 @ 2.0 V	0.12	TUTWSN [108]

The most commonly used MCUs have remained the same in WSN research during the last decade. In [190], out off 40 surveyed WSN research deployments, the 8-bit Atmel ATMega128 series was used in 16 and 16-bit Texas Instruments MSP430 series in 14 deployments. These MCUs have been selected for resource constrained WSNs due to their low price, low energy consumption, and small size.

WSN platforms with more computing resources exist, such as ARM Cortex-M3 equipped Preon32 [205], XScale 32 bit CPU equipped IMote [94], and an ARM11 equipped general purpose computer Raspberry Pi [171]. However, the energy consumption of these platforms is too high for long-term operation using small batteries. For example, Preon32 [205] has an active current consumption of 3.7 mA at 8 MHz and 1.3 mA in sleep mode, compared to the active 1.0 mA at 4 Mhz and 120 nA sleep current consumption of PIC18LF8722 [148].

The recently emerged ARM Cortex-M0+ MCUs provide 32-bit computing power with similar specifications to 8-bit MCUs. For example, STMicroelectronics provides a Cortex-M0+ MCU STM32L062K8 [189] that has 64 KB of program memory, 8 KB of Random Access Memory (RAM), 165 μA at 1 MHz 1.2 V run mode and 655 nA deepest sleep mode current consumption. These MCUs will allow more computing power on battery-powered WSN nodes, but the memory resources are not significantly larger.

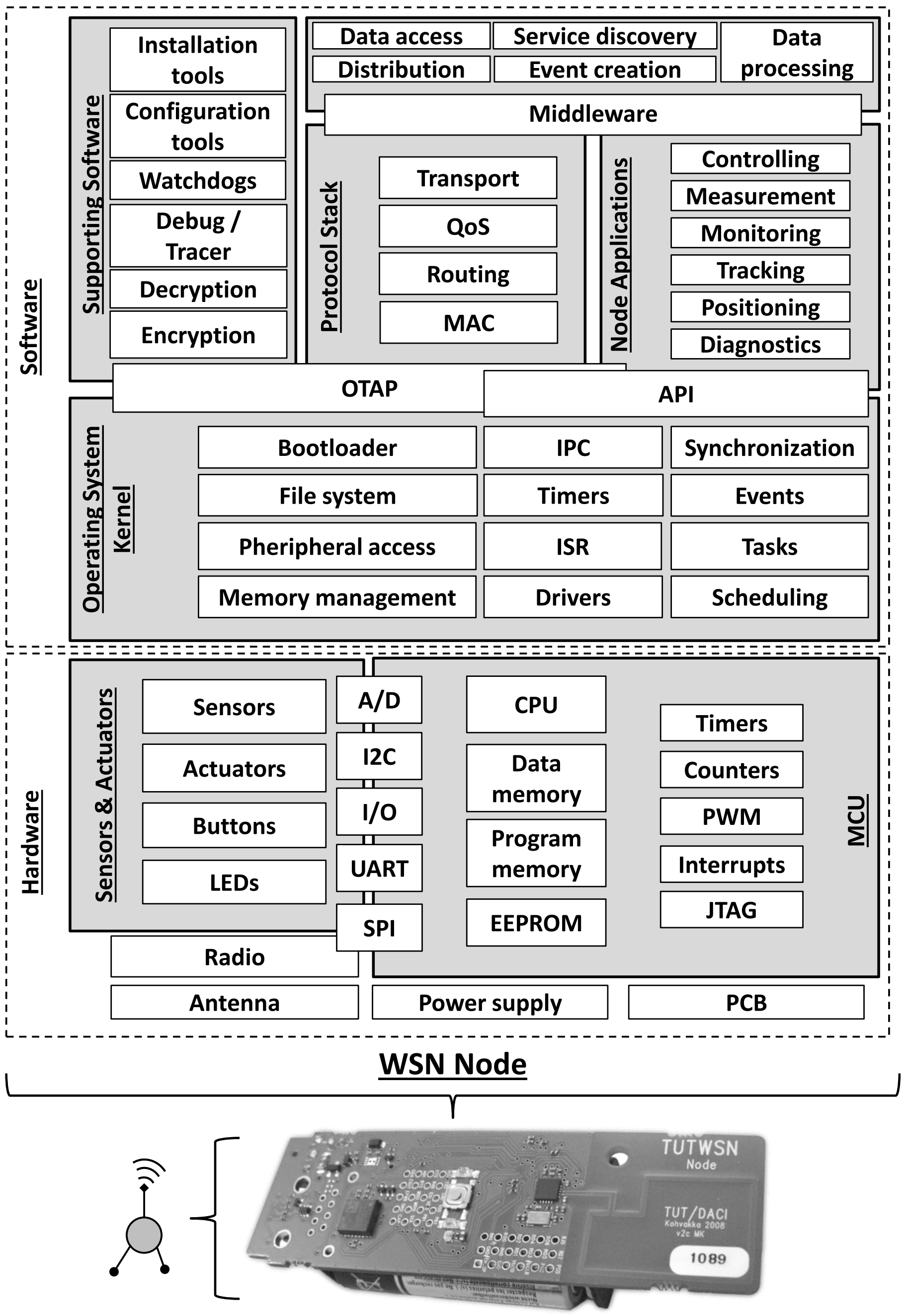


Fig. 5: Typical HW and software components of a WSN node. A TUTWSN node is used as an example of WSN node platforms.

The transistor density advances in IC technology will allow physically even smaller resource constrained nodes in the future. These nodes can be used in new applications. Eventually, the nodes will achieve the envisioned dust size [212], and a subset of the WSN nodes will remain resources constrained, hence, the results of this thesis will remain topical.

2.2.2 Radio Communication

Active radio communication HW consumes multiple times more energy than active MCU, as can be seen from the measurements of [P2]. To achieve low energy consumption, the communication HW usage is optimized with a WSN-specific Medium Access Control (MAC) protocol [2, 3, 49]. For example, communication HW active time is minimized using a Time-Division Multiple Access (TDMA) MAC protocol that tightly synchronizes the communication into low duty cycle frames with a short communication time [108]. The routing is typically a multi-hop mesh that avoids single points of failure and distributes the packet load over several hops. The nodes relay data hop-by-hop toward the collection points [190, 192], where each hop adds a delay to the packet delivery. Therefore, WSN communication set limitations on the data sending intervals, data packet sizes, and data delivery delays, which all affect abstraction design on the WSN nodes.

The radios used on the WSN node platforms are presented in Table 2. The throughput is constrained by low duty cycle communication. If the radio throughput is 250 kbit/s while the low duty cycle communication uses one 10 ms frame a second between two nodes, the maximum throughput of that hop is 2.5 kbits/s. This would not be enough to transfer real-time speech, as it requires a minimum of 4 kbit/s throughput. Payloads of 10–30 B are typically used in WSN deployments [190].

2.3 Node Abstractions

A WSN OS is an implementation of the node abstraction. An OS consists of a WSN OS kernel that executes the tasks of a protocol stack, an OTAP, middleware, supporting software, and WSN applications as depicted in Figure 6. The tasks create the functionality of the WSN and the WSN OS kernel schedules the tasks according to its design. The tasks require additional components from the kernel, such as events, Inter-Process Communication (IPC), synchronization, and memory management. The kernel, Interrupt Service Routine (ISR), peripheral drivers, timers, and file

Table 2: *Radios used in the existing resource constrained nodes. Range is an estimate of the maximum possible.*

Radio	Freq. (MHz)	Throughput (kbit/s)	Range (m)	Platforms
XBee-ZB [50]	2400	250	120	Arduino [11] Libelium Waspmote [129]
XBee-802.15.4 [50]	2400	250	90	“
XBee-900 [50]	900	156	10000	“
XBee-868 [50]	868	24	10000	“
Nordic nRF24L01 [159]	2400	2000	180	TUTWSN [108] Arduino [11]
Nordic nRF905 [160]	433	50	500	TUTWSN [108]
Atmel RF230 [17]	2400	50	500	MEMSIC IRIS [143]
Texas Instruments CC2420 [200]	2400	250	50	MEMSIC MICAz [144] MEMSIC TelosB [145] Shimmer [172]
Texas Instruments CC1101 [199]	315-915	600	2000	TUTWSN [108]

system [203] are abstracted behind one WSN OS Application Programming Interface (API) [190].

A review of WSN OS kernels, programming models, and OTAP is given in the following sections. The supporting software of WSN OS is left out of this thesis, since the abstraction methods are traditional interfaces, such as file systems [203], and dynamic memory allocation [132].

2.3.1 WSN OS Kernels

Research on WSN OSs has proposed two different approaches for multitasking [48, 51, 190]: cooperative event-driven kernels and pre-emptive multithreading kernels. The following sections cover and compare these kernels.

A *cooperative event-driven kernel* executes an event-handler after an associated event has occurred [48, 58, 115, 139]. Because of the cooperation, event-handlers must wait until the currently running event-handler willingly yields or ends its execution. The only exception is the event-handlers executed in interrupt handlers, but they are limited to a few specific tasks.

Cooperative event-driven kernels are proposed for resource constrained WSNs due to their small memory footprint. The kernel implementation is small in size, one event-handler requires only a few bytes of data memory, and switching event-handlers is

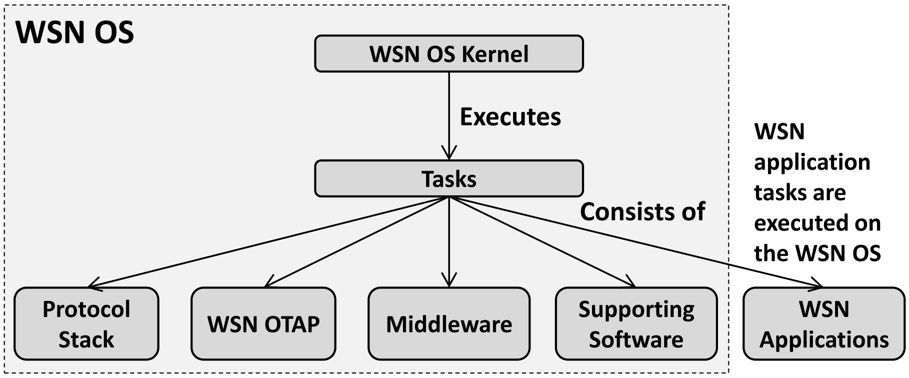


Fig. 6: *WSN OS definition as a node abstraction.*

fast and has a low overhead operation similar to a function call. As a result, one node can execute hundreds of event-handlers.

The down side of cooperative event-driven kernels is the challenging task programming approach [59, 115, 190, 191, 213], where event-handlers must either quickly complete or voluntarily yield the execution to avoid exhausting other event-handlers from the execution. As a result, combining a high timing accuracy task and a long-running task is challenging. Furthermore, sporadic event-handlers must wait for the completion of the currently executing event-handler, which can result in problems in WSN applications that require a fast reaction to outside events. These complexities must be taken into account by the application programmer.

Pre-emptive multithreading kernels provide threads for multitasking. The threads are forcibly removed (pre-empted) by the kernel to give execution time for other threads [56, 213]. The threads can be programmed without realizing the execution needs of other threads running in the system. The pre-emptive kernels are proposed for those WSNs that require high accuracy timing or a fast reaction to events [115].

Thread programming is familiar to developers from desktop computer programming [59]. As a drawback, each thread requires a stack located in the data memory, where the context of the execution environment is stored. Context storing and restoring during the pre-emption requires more execution cycles than changing an event-handler [56, 213]. Also, the pre-emptive kernels have a larger and more complex implementation compared to event-driven kernels due to the context switching.

Related Research on WSN Operating Systems

Related research on WSN OSs is summarized in Table 3. The table includes the WSN OSs proposed for the resource constrained WSN platforms.

TinyOS [84] and Contiki [58] are event-driven kernels. Contiki provides protothreads [59] as a solution for complex event-handler programming. Protothreads are explained in detail in the next section; however, protothreads do not solve the timing accuracy problem of the event-handler programming. MANTIS OS [24] and Nano-RK [66] are pre-emptive kernels that suffer from high overheads. LIMOS [223, 226] is a hybrid that provides pre-emption inside cooperative event-handlers. However, the pre-empted parts are scheduled cooperatively, thus LIMOS does not solve the large memory overhead of the pre-emptive kernels.

The remaining related works concentrate on different OS design issues rather than kernel types. For example, t-kernel [77] concentrates on task execution security by implementing efficient virtual memory mimicking and RETOS [36] achieves the same behavior with compile-time modification and run time checking. Task security ensures that a task does not violate the data of other tasks, but this does not affect the problems of event-driven or pre-emptive kernels.

Conclusions on WSN OS Kernels

The benefits and drawbacks of both kernel types are summarized in Table 4. Neither kernel type fulfills the specific needs of WSN application development [190, 213]. The event-driven kernel lacks timing accuracy and imposes a challenging programming approach. The pre-emptive kernel has high overheads that may limit the number of concurrent tasks, and thus complicate the development.

2.3.2 WSN Programming Models

WSN programming models can be divided into low-level and high-level methods [154, 168, 176, 191]. The high-level methods create applications over the whole network, for example using Structured Query Language (SQL) middleware presented in Section 2.4.1. The low-level methods consist of two main approaches: OS API programming and virtual machine programming.

The WSN OS API is used with a programming language, typically C, to construct

Table 3: *Related WSN OS proposals.*

Operating System	Type	Design features
TinyOS [84]	Event-driven	One of the first WSN kernels, open source, application development with NesC [72], thread and code protection extension [9].
MANTIS OS [24]	Pre-emptive	One of the first pre-emptive kernels.
Contiki [58]	Event-driven	Dynamic loading, multithreading library, open source, event-driven programming with protothreads [59].
SOS [80]	Event-driven	Dynamic module loading and primitive module execution safety.
Nano-RK [66]	Pre-emptive	High timing accuracy and deadline guarantees.
t-kernel [77]	Pre-emptive	Small overhead virtual memory and memory protection.
RETOS [36]	Pre-emptive	Dynamically reconfigurable and user/kernel space separation with software.
Pixie OS [133]	TinyOS extension	Resource aware dataflow programming model, where task reserved tickets follow availability and reservation of the resources.
LiteOS [33]	Pre-emptive	UNIX -like file system approach. Each node can be accessed with terminal connection.
CORMOS [218]	Event-driven	Communication oriented design.
TMO-NanoQ+ [220]	Pre-emptive	A time and message triggered pre-emptive task scheduling. Also, supports cooperative compile-time serialization scheduling.
SenOS [107]	State machine	A finite state machine based OS where state machine applications are described as sequence of actions.
OSone [167]	Event-driven	Thin hierarchical distributed OS that abstracts a distributed WSN to a one processing computer.
LIMOS [223, 226]	Hybrid	Each cooperative event-handler can execute several pre-emptive threads.
FreeRTOS [71]	Pre-emptive	General purpose open source Real-time Operating System (RTOS), small memory overhead compared to other general purpose RTOSs.
SensorOS [115]	Pre-emptive	High timing accuracy.

WSN applications. NesC [72] and protothreads [59] are specially designed programming methods for WSNs.

Protothreads [59] are a state machine abstraction for event-driven kernels that were first used in Contiki [58]. Protothreads are implemented with C precompiler macros that provide a pre-emptive thread like API for event-handlers. As protothreads have a special approach to traditional C programming, Listing 2.1 presents an illustrative example of timer event usage with protothreads. The actual program code of the protothread commands *PT_<command>* is inlined after the precompilation. This

Table 4: *Comparison of the kernel types.*

Kernel Type	Benefits	Drawbacks
Pre-emptive kernel	Guaranteed timing accuracy. Easy programming model for combining timing critical tasks and long running tasks. Familiar programming model.	High memory overhead when deployed. Pre-emption execution overhead increases energy consumption.
Event-driven kernel	Low memory and execution overhead.	Challenging to combine timing critical tasks and long running tasks. Non-familiar programming model for application developers.

complicates run time debugging, since lines 3, 6, and 9 of Listing 2.1 contain code that the debugger cannot break to. Protothreads simplify splitting the tasks in the event-handlers, but they do not solve the high accuracy timing issues.

NesC [72] is an event-handler programming language for TinyOS. The NesC code is a dialect of C programming language and is compiled into a full C programming code. Thus, NesC requires the application programmer to adopt a new language. As a similar illustrative example to that for protothread, Listing 2.2 presents timer event usage with NesC. Interface declarations are deprecated for the sake of presentation clarity. The code contains an initialization of the event-handler in lines 2–5, setting up a timer in lines 7–13, and the actual event-handling code in lines 15–18.

A Virtual Machine (VM) typically runs on top of a small OS kernel and a protocol stack. VM applications are developed using an HW-independent byte code that separates the application from the running HW. This increases heterogeneity and portability, since the same VM byte code runs on different MCUs without any changes. As a downside, VM approaches have a higher execution and memory overhead than running native machine code. Also, VMs require handling a new byte code from the programmer, and may limit the application development, e.g. only allow access to predefined events.

Maté [124] is a VM for TinyOS that is a stack computer with predefined event triggers, and built-in sampling, sending, and receiving instructions. As a similar illustrative example, Listing 2.3 presents sensor sampling and sending with Maté. Darjeeling [30] is similar to Maté. Impala [131], SensorWare [28], and MagnetOS [20] are commonly cited VMs in WSN research, but they are too resource-consuming for re-

```
struct etimer timer;
PT_THREAD(example(struct pt *pt))
{
  PT_BEGIN(pt);
  while(1) {
      etimer_set(&timer, 1000);
      PT_WAIT_UNTIL(pt, PROCESS_EVENT_TIMER);
      // Application logic.
  }
  PT_END(pt);
}
```

Listing 2.1: *Initiation and handling of a timer event with protothreads in Contiki*

```
implementation {
    command result_t StdControl.init() {
        return SUCCESS;
    }

    command result_t StdControl.start() {
        return call Timer.start(TIMER_REPEAT, 1000);
    }

    command result_t StdControl.stop() {
        return call Timer.stop();
    }

    event result_t Timer.fired() {
        // Application logic.
        return SUCCESS;
    }
}
```

Listing 2.2: *Initiation and handling of a timer event with NesC*

source constrained WSNs. MagnetOS [20] is a general purpose Java VM distributed over ad hoc nodes, SensorWare [28] implementation takes 240 KB of program memory, and Impala [131] is implemented for Personal Digital Assistants (PDAs).

Protothreads and NesC simplify event-handler programming, but they do not solve the timing and long running task issues. NesC and the VMs require programmers to adopt a new programming language and programming model for the applications. Compared with native C, a VM can restrict application development.

```
pushc 1 // Set predefined sensor ID parameter to the stack
sense   // Sample the sensor using the built-in instruction
pushm   // Push message to the stack
add     // Add sampled value to the message
send    // Send the message
```

Listing 2.3: *A sensor sampling with Maté.*

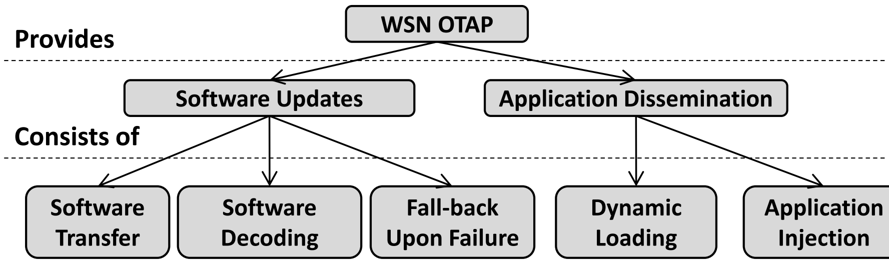


Fig. 7: *WSN OTAP definition.*

2.3.3 Over-the-Air-Programming

OTAP is a method for fixing software errors, adding new functionality, or adding new applications to a WSN without physical access to the nodes. WSN OTAP is required, since the network may consist of thousands of devices, which makes physical programming impractical, or the network may be deployed to an environment that is not accessible [31].

Figure 7 presents the definition for WSN OTAP. A WSN OTAP provides software updating and application dissemination methods. Software updating consists of transferring the software, decoding the software on the node, and a fall-back procedure in a case of failed update. Application dissemination consists of injecting new applications into the network and dynamically loading the new application in to the execution. The entire set of software on a node is typically referred to as *firmware*.

Five OTAP methods can be distinguished: a VM, a loadable library, a firmware dissemination, an incremental dissemination, and rateless codes. Table 5 provides the benefits, drawbacks, and existing proposals for these OTAP methods. The methods are described in the following paragraphs.

Since VMs separate the executed software from the HW, the VM byte code can be updated and relocated without any modifications to the actual embedded software. The new VM byte code is disseminated to the nodes using the protocol stack and the nodes take it to the execution. If the VM byte code runs on top of a kernel and a protocol stack, the VM method can only fix, update, and add new WSN applications. The protocol stack or the OS cannot be updated or fixed [179], unless additional updating methods are implemented. For example, VM* uses incremental updating for the system software and VM itself [109].

In the loadable library method, the OS dynamically loads parts of the native firmware. The libraries can be updated by disseminating them separately to the network [54,57, 58,140]. This method has a high overhead: the loading is a complex operation and requires additional memory space. The relocation of the new code is execution time- and energy-consuming. This method only allows updating of the libraries, which must be preselected. In addition, implementing the dynamic loading may require specialized development tools, such as scripts to modify the compiled firmware as location-independent.

In firmware dissemination, a large part or all of the firmware is disseminated to the network and nodes load the new firmware either onto an external flash memory or directly to the program memory [88,89,165]. The dissemination can be over the WSN protocol stack or a specialized dissemination protocol. The firmware dissemination is the most capable method since it can update and add new features to the entire firmware. However, heterogeneity support is inefficient. If the HW has a high level of heterogeneity, the firmware must contain all of the software for each HW configuration, or different firmware has to be disseminated to differently configured nodes. As a result, the dissemination of small fixes or new applications that modify a small part of the code is a resource-consuming operation.

Incremental dissemination is a compression method for firmware dissemination. Only a delta file is disseminated, which describes how the existing firmware must be modified to achieve the new firmware [53,55,86,97,164]. The delta file contains instructions to relocate and delete the existing code, and additions of new parts that do not exist on the current firmware. Because of the HW-specific implementation, the compression of incremental updating is better than with traditional compression algorithms [55,202].

The incremental update is an efficient method when the modifications are small. As a down-side, firmware reconstruction can be a time- and energy-consuming task and requires architecture-specific implementations, since function calls and variable addresses need to be modified on relocation. Also, if large part of the firmware change, the benefits compared to firmware dissemination are lost.

Table 5: *Comparison of the OTAP methods.*

OTAP method	Purpose	Benefits	Drawbacks	Existing proposals
VM	Update application software.	The software is independent from the HW and easily disseminated.	High overhead. May restrict updates to application code only, e.g cannot fix issues in the VM itself.	Maté [124], VM* [109], Impala [131], SensorWare [28], and Darjeeling [30]
Loadable library	Update parts of the firmware and the application software.	Only selected parts of the firmware needs to be disseminated.	High implementation and execution overhead. Requires support from the implementation tools.	ELON [54], FlexCup [140], Contiki dynamic library [57, 58]
Firmware dissemination	Update the entire firmware.	Small execution overhead. Low energy overhead if updates are rare.	Restricted heterogeneity support. Careful design required to avoid network fragmentation during the update.	Deluge and Deluge2.0 [88, 89], Stream [165]
Incremental dissemination	Update modified parts of the entire firmware.	Small updates require small amount of data transfer and are done efficiently.	On large updates, efficiency degrades due to the parsing a new firmware. Complex implementation.	Rsync [97], RMTD [86], Zephyr [164], R2 [53], R3 [55]
Rateless codes	Improve efficiency of the dissemination of the entire firmware.	High immunity to packet loss. Reduces packet transmission.	Increased overhead on new software decoding from the encoded codes.	ReXOR [52], RatelessDeluge [79], Synapse [178], Synapse++ [177]

The rateless code method encodes the firmware before sending. The code stream can experience a certain level of packet loss without the need for retransmission: the receiver can patch missing packets on its own [52, 79, 177, 178]. This reduces the amount of disseminated data and increases the update success rate in the unreliable network conditions of WSNs. As with incremental updates, rateless codes have high execution, program memory, and data memory overheads, due to the required decoding at the receiving node.

In conclusion, none of the presented OTAP methods completely match both requirements of software fixes and application dissemination. VMs add overhead and do not allow software fixes for non-VM code. The same is true of loadable libraries. Firmware dissemination is inefficient with small updates and application dissemination. These issues are solved by incremental updating, but it adds overhead and does not solve heterogeneity issues. Rateless codes only improve dissemination, with a cost of execution overhead.

2.4 Network Abstractions

With a network abstraction, WSN applications are developed over the distributed nodes. The network abstraction is implemented using middleware [37, 78, 176, 211]. Middleware hides resource management, network management, and topology from the WSN application: the WSN application has methods to discover and access the services of the network without actual understanding how the network is organized. Services are typically measurements, actuators, and processing. Processing includes services such as data aggregation, data fusion, and event creation. Security and Quality of Service (QoS) can be integrated into middleware [78, 151, 211].

2.4.1 WSN middleware

WSN middleware research has been versatile for resource constrained WSN. Four common methods are found in the related research [37, 78, 83, 116, 141, 151, 154, 176, 179, 191, 211]: a WSN API, WSN as a database, shared memory through a tuple-space, and mobile agents [176]. The following section presents the methods targeted to resource constrained WSN nodes. The surveys typically cite Milan [156] and SensorWare [28], but these were excluded from this thesis since their approach is not designed for resource constrained WSNs.

WSN APIs have methods to discover services and deliver service data to interested

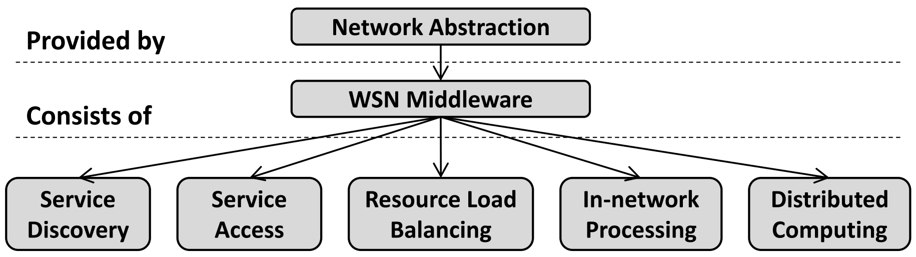


Fig. 8: *WSN middleware definition.*

```
SELECT AVG(temperature), nodeid, FROM sensors
  WHERE floor = 1
  EPOCH DURATION 60s
```

Listing 2.4: *An example of TinyDB data query for acquiring averaged temperature from Floor 1.*

parties [101, 187]. For service discovery, the network abstraction can advertise the available services [187] or respond to an injected request with the available services [101]. When the service is discovered, a subscription is made at the service provider, which then delivers the service data. The subscription describes a data delivery method that can be at an interval, on change, or a one-time delivery.

WSN as a database uses a query language for data acquisition, aggregation, and processing. TinyDB [136] and COUGAR [219] are well-known database proposals that use an SQL-like query language. Listing 2.4 gives an illustrative example of TinyDB data acquisition for comparison to low-level OS programming in Section 2.3.2. The abstraction is higher, as the executed code on the nodes is not shown to the application programmer, and the query concentrates purely on data acquisition. WSN as a database typically allows the aggregation of data and execution of simple processing. For example, an alarm can be generated when a set limit is exceeded.

TinyDB has a static set of available sensors and processing attributes that the application programmer may use. TinySOA [173, 174] uses dynamic services as the abstraction, where the WSN nodes publish their services when joining the network.

In *tuple-spaces*, data is shared between a group of nodes through shared memory, where an application can store and retrieve data as tuples. Tuple space originates from Linda [73], which is a shared memory proposal for traditional distributed systems. A tuple is a combination of data points, e.g. *(<node ID>; <data type>; <data>)*. Figure 9 depicts WSN tuple space middleware in action. A temperature application

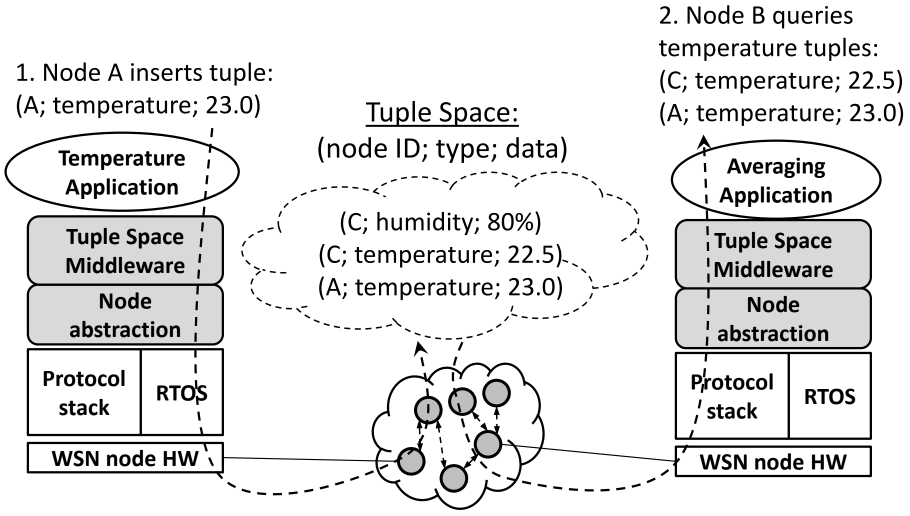


Fig. 9: *A tuple space middleware and its functioning on top of a node abstraction.*

on *Node A* inserts a tuple into the tuple space. *Node B* queries temperature tuples from the tuple space for averaging, and receives tuples from *Node A* and *Node C*.

TinyLIME [47] and TeenyLIME [45] are tuple space middleware for WSNs. Both are based on LIME [157] middleware for ad hoc networks, which proposes mobile agents with a tuple space. TinyLIME uses resource constrained WSN nodes as data sources in a tuple space for mobile sinks of higher computing power. The sinks contain fixed agents that implement the application logic. TeenyLIME works completely on WSN nodes. The node implementations of TinyLIME and TeenyLIME build on top of TinyOS.

TinyLIME [47] uses *fixed agents* as an application logic implementer. An agent activates itself in the execution when the desired condition occurs, e.g. a tuple required by the agent is received. *Mobile agents* migrate in the network by cloning themselves from node-to-node according to given behavior conditions. They implement the distributed application behavior in the network by storing their internal state over migration [211]. Figure 10 presents mobile agent middleware in a WSN, where an example mobile agent travels according to a temperature limit. For example, such a mobile agent could map the progression of a forest fire and guide firefighters. Agilla [69] is WSN middleware that uses mobile agents together with a tuple space.

Table 6 presents a summary of WSN middleware and their key features. These WSN middleware and high-level programing approaches are well covered by the existing surveys [37, 78, 83, 116, 141, 151, 154, 176, 179, 191, 211].

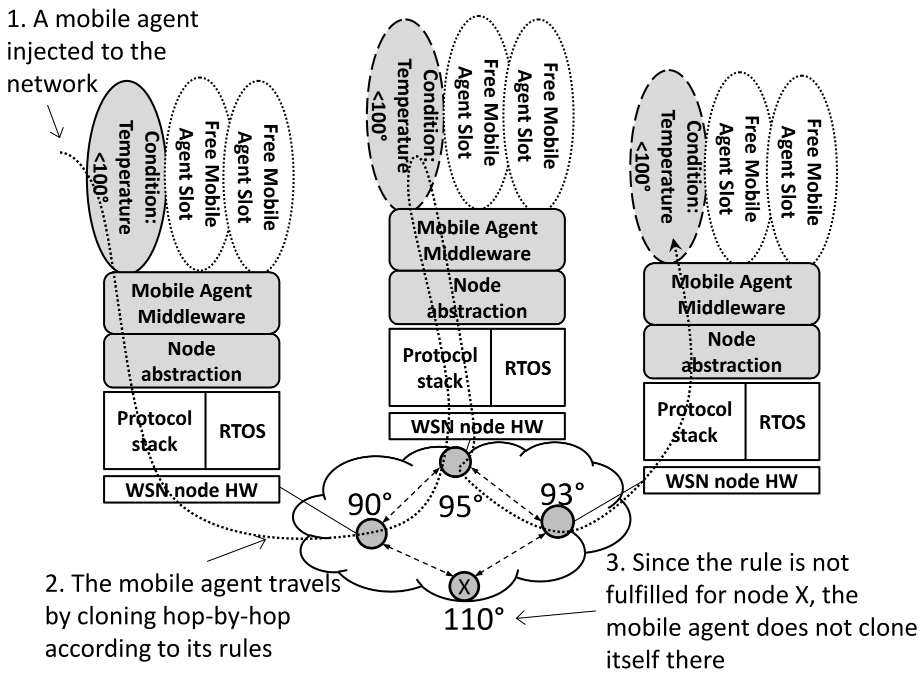

1. A mobile agent injected to the network
Condition: Temperature <100°
Free Mobile Agent Slot
Free Mobile Agent Slot
Mobile Agent Middleware
Node abstraction
Protocol stack
RTOS
WSN node HW
90°
95°
93°
110°
X
2. The mobile agent travels by cloning hop-by-hop according to its rules
3. Since the rule is not fulfilled for node X, the mobile agent does not clone itself there

Fig. 10: *A mobile agent middleware and its functioning on top of a node abstraction.*

In conclusion, WSN middleware has a high variation between different approaches and there are no directly comparable qualitative metrics for them. According to [154, 211], the variety is due to the specific application scenarios to which the middleware are targeted. Also, lack of collaboration possibilities between different middleware is seen as a problem. Database, tuple space, and mobile agent middleware are considered theoretical approaches, as they do not directly fit the real-world application requirements of sense-and-react applications [154]. Therefore, they are not widely adopted in WSN field experiments [190].

The current WSN middleware may hide heterogeneity inside one WSN, but WSN middleware do not work over different WSNs and other sensing technologies that WSN application development requires [83, 179, 191, 211]. More sophisticated processing is expected in the future, where the processing will take place at the various levels of the abstractions [37, 83].

Table 6: *Summary of the WSN middleware for resource constrained WSNs.*

Middleware	Type	Specialty
Mires [187]	Interface	Publish/subscribe data delivery, advertised services, and in-network aggregation services.
WSN API [101]	Interface	Service discovery through injected requests, and data delivery through queries.
TinyDB [136]	SQL Query	A database middleware for TinyOS that has in-network aggregation and processing support [135].
Cougar [219]	SQL Query	One of the first SQL like middleware for WSNs.
SINA [181]	SQL Query	Provides integrated scripting language for in-network applications.
DSWare [127, 128]	SQL Query	Adds a confidence function for fused events. Reduces communication with in-network processing.
TinySOA [173, 174]	Service Query	Abstracts node resources as dynamic services. Provides event and condition query language to activate the services.
TinyLIME [47]	Shared tuple space	Nodes deliver data to the tuple space of mobile sinks. Fixed agents react to the available WSN data in the tuple space, which provides service discovery for the sinks.
TeenyLIME [45]	Shared tuple space	Creates a shared memory over the WSN nodes.
Agilla [69]	Shared tuple space	Mobile agents clone and migrate from node to node.

2.4.2 In-Network Processing

In-network processing of a WSN network abstraction can improve energy efficiency, bandwidth usage, and reliability. For example, nodes in a room could deliver temperature packets to one node that calculates an average, drops the rest of the packets, and only sends the averaged packet forward. This would reduce packets on the routing path outside the room. In-network processing is presented in this thesis as it works as one motivation for the embedded cloud.

Krishnamachari et al. [112] used mathematical and simulation analysis on data aggregation models. The result proposed 50%-80% energy savings with WSNs. Prakash et al. [170] used real HW parameters with simulation models and managed to reduce energy consumption by 74% from 270 mJ to 70 mJ. TinyDB, TAG, and SKETCH queries with in-network aggregation were tested in [134] by Luo et al. The results suggest that aggregation can improve energy efficiency and data quality, or reduce packet loss.

Proportional-Integral-Derivative (PID) controlling and event detection use cases were simulated in [8] and the proposed in-network processing reduced traffic, delay, and energy consumption by up to 85% depending on the topology when compared with processing on external logic. In-network processing increases reliability and responsiveness in controlling applications, since the controlling node can make adjustment decisions without a connection to the external logic.

In-network processing consumes CPU execution time and the packet exchanging can increase if the processing does not fit the routing topology or the physical distribution of the nodes [134, 170]. In addition, in-network processing can reduce data quality or accuracy when reducing number of packets. Thus, using in-network processing is not always beneficial: a method for balancing overheads and benefits is required. Such methods are not reported in the related works.

3. RESULTS FOR NODE AND NETWORK ABSTRACTIONS FOR WSN NODES

This section summarizes the contribution of this thesis to research on node and network abstractions on resource constrained WSN nodes. The contributions are as follows.

- HybridKernel solves the overhead problem of pre-emptive kernels and challenging event-handler programming with a scalable hybrid compromise.
- A combination of Program Image Distribution Protocol (PIDP) and Process Description Language (PDL) form an OTAP method that efficiently updates all of the software and allows dissemination of applications over heterogeneous platforms. PIDP is a firmware dissemination method, and PDL is a byte code VM that uses abstracted services of sensing devices as operation parameters.
- Distributed Middleware (DiMiWa) and PDL provide network abstraction over heterogeneous WSNs. As they are key components of the embedded cloud, they are presented in detail with the embedded cloud in Chapter 5.

3.1 WSN Node Platform Used on the Thesis

The proofs of concepts in this thesis are implemented for the Tampere University of Technology Wireless Sensor Network (TUTWSN) nodes [108]. They consist of PIC18F8722 MCU [148] with a selection of 2.4 GHz, 833Mhz, and 433 MHZ radio chips. The TUTWSN nodes are typically powered with two AA batteries. Compared with other WSN node platforms, the PIC18F8722 MCU has similar performance metrics to Atmel ATmega128L as shown in Table 1. The TUTWSN nodes compare to the MEMSIC MICAz [144] and MICA2 [146] node platforms that are frequently used in WSN research [190].

TUTWSN has low energy and low latency protocol stacks available [104, 108, 192]. Low energy TUTWSN is a multi hop ad hoc mesh WSN that functions for 1–5 years with two AA batteries while delivering measurements from 30 seconds - 5 minutes intervals. Low latency TUTWSN is a multi hop ad hoc WSN that has a latency under 1 second per hop, mains powered routing nodes, and battery powered mobile nodes.

3.2 HybridKernel

Cooperative event-driven kernels and pre-emptive kernels are not completely satisfactory for WSN application development. Event-driven kernels have an unfamiliar programming approach and are not suitable for combining long-running and high timing accuracy tasks. Pre-emptive kernels have high memory overhead due to the context stacks required that limit the amount of usable tasks on resource constrained WSN platforms.

HybridKernel [P1] consists of a pre-emptive base and a cooperative event-driven extension. HybridKernel ensures high timing accuracy between pre-empted tasks, or *processes*. The event-driven extension enables multiple low memory overhead tasks, or *threads*, within one pre-emptive process, which ensures scalable overheads. The threads are programmed using protothreads [59]. The threads have their own event waiting and yield kernel API functions. The rest of the kernel API is shared between the base and the extension consisting of events, timers, an IPC method, a mutual exclusion synchronization, and memory management. As a result, high timing accuracy tasks and long-running tasks can be programmed for HybridKernel as on traditional OS through one homogenized API.

3.2.1 Comparison of Scheduling Methods

The differences in scheduling between the cooperative event-driven kernel, the preemptive kernel and the hybrid scheduling of HybridKernel are illustrated in Figures 11, 12, and 13. In this scheduling example, the protocol stack is a tightly synchronized periodic task that requires high timing accuracy. The temperature and humidity tasks are periodic measurement tasks. The motion detection is a measurement task that reacts to the sporadic events of a motion detection sensor. The new software decoding task is a long low priority task, such as an incremental updating task. This example is a typical WSN application scenario: as reported, WSN deployments typically use 2–5 OS tasks and 2–6 application tasks [190]. The time is compressed in these figures for the sake of clarity.

On the event-driven kernel in Figure 11, the other tasks need to yield from execution

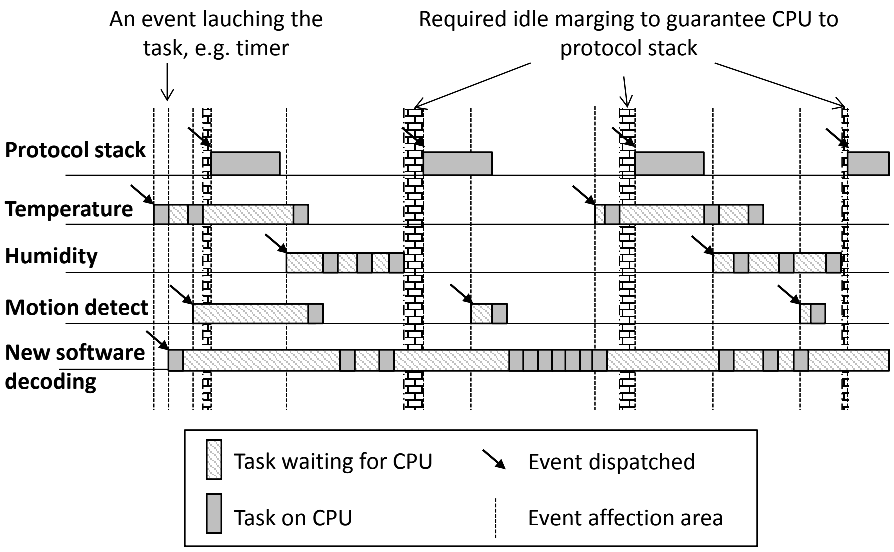


Fig. 11: *The scheduling example for an event-driven cooperative kernel. The tasks are scheduled in round robin fashion and the task must yield voluntarily. The tightly synchronized protocol stack requires that other tasks yield within a margin time.*

voluntarily within an idle margin time to guarantee CPU for the protocol stack. As a result, task completion times are delayed, CPU time is wasted during the brick wall idle margin, and the tasks need to be split by the programmer. The pre-emptive kernel in Figure 12 can forcibly take the protocol stack in to the execution when the time is due. As a result, other tasks complete faster, CPU time is not wasted, and the programmer does not need to split the tasks.

On HybridKernel in Figure 13, the temperature, humidity, and motion detection tasks are executed as threads in one process. The protocol stack, and the new software decoding tasks run on two additional processes. Compared with the event-driven kernel, HybridKernel reduces task completion times and wasted CPU time. Also, the measurement and new software decoding tasks can be implemented without realizing the timing requirements of the protocol stack. Compared with the pre-emptive kernel, HybridKernel requires 3 stacks instead of 5, and 14 context switches instead of 16 in this example.

3.2.2 Proof of Concept Implementation

The proof of concept of HybridKernel on the TUTWSN platform requires 9083 B of program memory, 89 B of data memory, achieves 1 μs timing accuracy for the

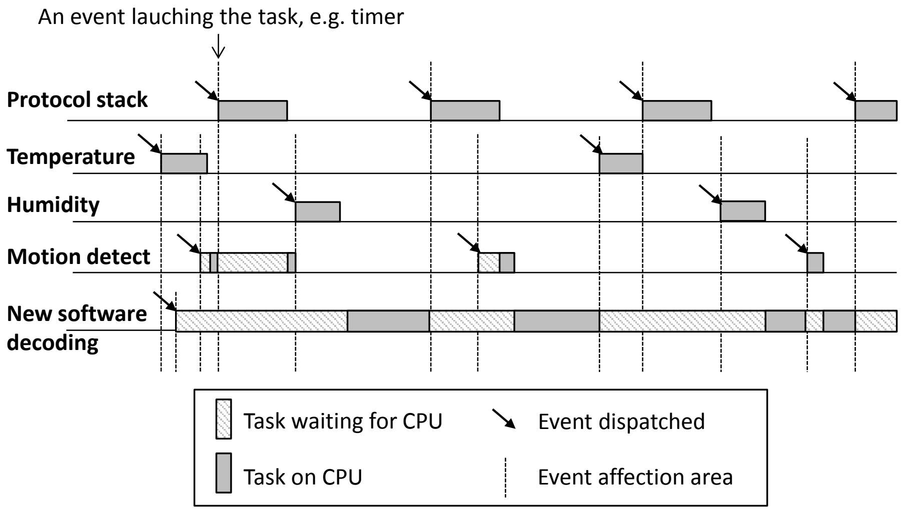


Fig. 12: *The scheduling example for a pre-emptive kernel. Margins are not needed since the kernel pre-empts lower priority threads. The temperature, humidity, and motion detect application tasks share the same priority.*

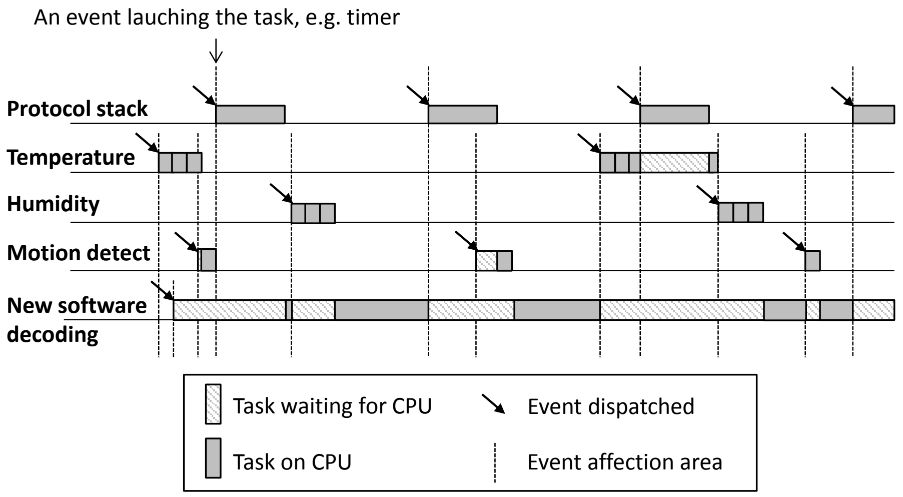


Fig. 13: *The scheduling example for HybridKernel. Margins are not needed since the kernel pre-empts lower priority threads. The temperature, humidity, and motion detect application tasks run on the same process, and require yielding voluntarily due to cooperative event-driven scheduling.*

highest priority process, and allows a scalable data memory overhead according to the application requirements. The process context switch takes 90 μs on average on a 4 MHz PIC18F8722 MCU measured with an oscilloscope. Thread switching takes a few μs: the accuracy was limited due to the speed of the measurement arrangement.

Table 7: A comparison of data memory consumption of the WSN OS kernels for five tasks with three levels of pre-emption. The values with * have been calculated from the available source code.

OS	Kernel (B)	A Pre-emptive Task (B)	Pre-emptive Tasks total (B)	Stacks (B)	An Event-handler (B)	Event-handlers total (B)	Total (B)
HybridKernel	89	28	84	384	31	93	566
SensorOS	115 [115]	17 [115]	85	640	-	-	840
TinyOS	178 [84]	43* [42]	129	384	46 [84]	138	829
Contiki	230 [58]	8* [40]	24	512 [1]	15* [41]	45	811
MANTIS	144 [24]	10 [24]	50	640	-	-	844

As a comparison, a TinyOS task posting takes 80 clock cycles [125], which would be 10 μs with a 4 MHz clock speed. SensorOS [115] is a pre-emptive kernel that requires 6964 B of program memory on a PIC18F8722. Thus, the event-driven extension of HybridKernel adds ca. 2 KB of program memory overhead.

Table 7 compares the data memory consumption of HybridKernel to four other WSN OSs with the use case of Figure 13. The values are taken from the publication or calculated for a PIC18F8277 MCU from the available source codes. A 128 B stack is used for each pre-emptive task and three levels of pre-emption are used for HybridKernel, Contiki, and TinyOS with TOSThreads [9]. Additional data memory consumption may be realized from events and other dynamic variables that the kernel and applications require to function. For HybridKernel, all necessary events that the kernel requires are included in the task memory consumption.

HybridKernel saves energy by reducing context switches when compared to pre-emptive only kernels. The energy consumption E_{total} of a kernel can be expressed as

$$E_{total} = P_a t_a + P_s t_s, \tag{1}$$

where P_a is the active MCU power consumption, t_a is the active MCU time during the observed time, P_s is the sleep power consumption and t_s is the sleep time. The sum of t_a and t_s give the observed time $t_{obs} = t_a + t_s$ which is the time-frame where the context switch reducing is observed. By eliminating active time t_a with the help of t_{obs}, the energy consumption E_{Pre} of a pre-emptive kernel can be presented as

$$E_{Pre} = P_a(t_{obs} - t_s) + P_s t_s = P_a t_{obs} + (P_s - P_a)t_s = P_a t_{obs} + \triangle P t_s, \tag{2}$$

[1] Contiki requires one additional stack for the kernel.

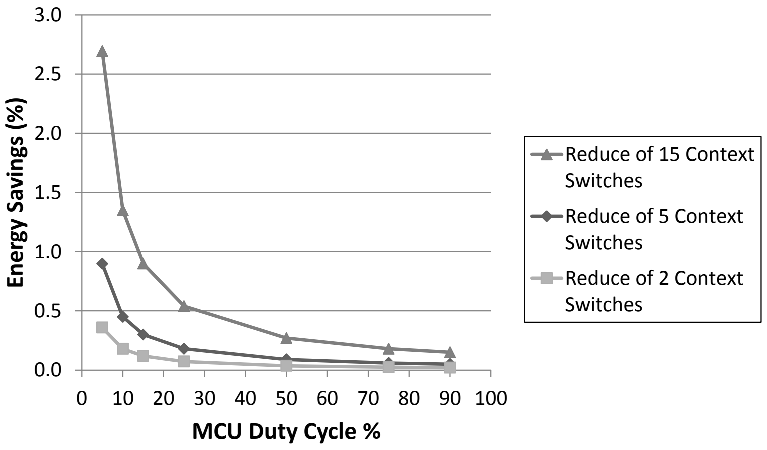


Fig. 14: *The energy savings of HybridKernel when reducing context switches of a pre-emptive only kernel. Duty cycle is the active to sleep time relation of the MCU. The plot is for 1 s observation time using PIC18F8722 context switch time and energy consumption.*

where $\triangle P$ is the difference $P_s - P_a$ in sleep and active power consumption.

The reduction in context switches increases the sleep time. Therefore, the energy consumption E_{Hybrid} of HybridKernel can be calculated by adding the reduced context switches to the sleep time of E_{Pre}, which results in

$$E_{Hybrid} = P_a t_{obs} + \triangle P(t_s + t_{cs}\triangle C), \tag{3}$$

where $\triangle C$ is the reduction in context switches and t_{cs} is the context switch time. Compared with a pre-emptive kernel, the energy savings of HybridKernel S_E can be calculated as a percentage:

$$S_E = 1 - \frac{E_{Hybrid}}{E_{Pre}} 100 = 1 - \frac{P_a t_{obs} + \triangle P(t_s + t_{cs}\triangle C)}{P_a t_{obs} + \triangle P t_s} 100. \tag{4}$$

Figure 14 plots the energy savings of Equation 4. The energy savings depend on the MCU duty cycle. If a pre-emptive kernel has a low duty cycle (short MCU active time and long MCU sleep time), the context switch overhead increases, as the context switches consume more of the active time. Therefore, HybridKernel is suitable for low duty cycling WSNs, where context switches occur over a short period of active time.

In addition to the results of this thesis and [P1], the implementation of a router node of low latency TUTWSN [104] was ported on HybridKernel in a research project [161]. The router nodes listened continuously for incoming packets. Compared with a non-interrupting loop kernel, HybridKernel port increased the packet receiving capability of the router nodes by reducing the packet handling delay. As a result, HybridKernel port improved low latency operation. In addition, HybridKernel port reduced code complexity. Respectively, it increased both data and program memory consumption. The increase in the data memory consumption was 477 B that was 12% of the available data memory. Energy consumption was not considered, since the low latency TUTWSN routers are mains powered due to the continuous listening.

3.2.3 Discussion of the Results

The requirement for hybrid design has been discussed in the literature. Strazdins et al. [190] reasoned that both cooperative and pre-emptive scheduling should be supported by the WSN OS according to the reported WSN deployments. Watfa and Moubarak discussed the same requirement in [213]. Both articles were published after the publication of HybridKernel.

HEROS [132] has a similar hybrid design as HybridKernel, but was published five years later in 2014. TinyOS and Contiki have external libraries for pre-emptive tasks. These libraries are not integrated into the kernel design, which requires special API functions that require attention from the application programmer. LIMOS is a hybrid kernel, but it conducts pre-emption inside cooperative event-handlers, which does not reduce overheads or improve timing accuracy between the event-handlers.

3.3 OTAP Method

An OTAP method is needed to fix software errors and add new features to OSs, protocol stacks, applications, and other WSN software when the WSN is deployed. VMs and loadable library methods only allow the applications and predefined software components to be updated. In addition, both methods add execution overhead. Firmware dissemination methods are not suitable for small updates and new application dissemination, which is solved by incremental dissemination methods. However, they have complex and device specific implementation, add overhead, and degrade in efficiency on large updates. Rateless codes can be used to reduce transmitted data, but they add execution overhead and do not provide any new methods.

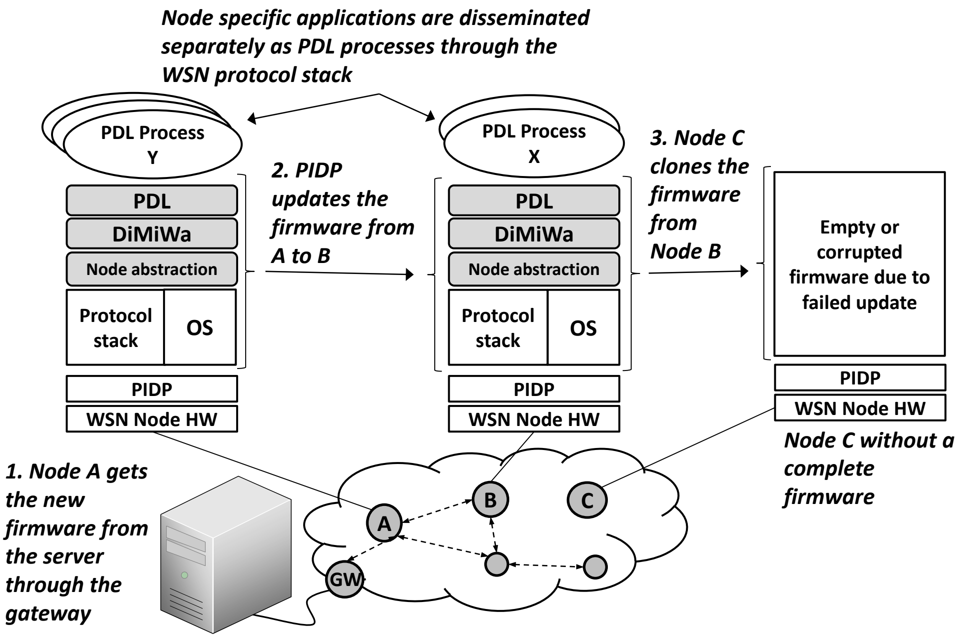

Node specific applications are disseminated separately as PDL processes through the WSN protocol stack
PDL Process Y
PDL
DiMiWa
Node abstraction
Protocol stack
OS
PIDP
WSN Node HW
2. PIDP updates the firmware from A to B
PDL Process X
PDL
DiMiWa
Node abstraction
Protocol stack
OS
PIDP
WSN Node HW
3. Node C clones the firmware from Node B
Empty or corrupted firmware due to failed update
PIDP
WSN Node HW
Node C without a complete firmware
1. Node A gets the new firmware from the server through the gateway
A
B
C
GW

Fig. 15: *The firmware OTAP and the application dissemination method with PIDP and PDL.*

The combination of PIDP and PDL forms a robust and energy efficient method for a WSN OTAP that allows both firmware updates and application dissemination as presented in Figure 15. PIDP allows the same firmware to be used for the whole WSN. The newly deployed nodes are automatically programmed with the newest firmware, which reduces maintenance and tracking of software versions. PDL allows node-specific applications, which can be dynamically and automatically updated. Both PIDP and PDL have been implemented for PIC18F8722 MCU and have been shown to feasible with proofs of concept in [P2] and [P6].

PIDP [P2] is a firmware dissemination method that clones all of the new software hop-by-hop from node to node. PIDP does not require fail-safe firmware on an energy-consuming external flash, since it has a robust fall-back method. Since PIDP is not part of the updated firmware, PIDP can always start a new transfer with a neighbor through an advertisement channel, if the updated firmware or the transfer fails. PIDP ensures that the whole network is eventually updated, even if some nodes are not within the reach of the network when the update starts. Furthermore, a new node without existing firmware can be deployed to the network and the PIDP will automatically clone the newest version from one of the neighbors.

Similar to other firmware dissemination protocols, PIDP is not well suited for application dissemination. The application software must be the same for of all the nodes, which wastes program memory and complicates the network configuration. Also, a separate configurator program is needed to define which applications can run on which nodes [P2 - P3]. Using PDL for the applications solves this issue. It is presented in detail in Chapter 5 and [P6]. PDL describes the application process as a byte code, and the PDL processes are disseminated using the WSN protocol stack, similar to the VM updating method. This method allows for unique application processes for each node, which can be added, updated, and removed separately of the WSN firmware updated by PIDP.

3.3.1 Proof of Concept Implementation

PIDP requires 6389 B of program memory and 22 B of data memory when implemented on a PIC18F8722. A receiving node consumes 1469 mJ of energy and 74 s of MCU time during one update on the TUTWSN WSN node platform at 4 MHz. A sending node consumes 1550 mJ of energy and 49 s of MCU time. The execution time difference is due to the verification of the transferred firmware. The energy consumption difference is due to the different radio listening times. If the advertisements can be done as part of the protocol stack signaling, as in TUTWSN, they do not increase energy consumption. As a result, one update on PIDP consumes under 0.01% of the available 20000 J energy. The PDL proof of concept implementation is covered in Chapter 5.

3.3.2 Discussion of the Results

The related OTAP methods are not efficient for both tasks of updating software and disseminating applications. This thesis shows that software updates can be efficiently and robustly done with PIDP. It does not require energy consuming external flash, updates all of the software, and has a robust fall-back method. To allow application dissemination, a separate abstraction of PDL provides an efficient solution for frequent application changes on heterogeneous nodes, as it works on top of the WSN stack. In conclusion, PIDP and PDL complete each other. PDL could be replaced with a VM approach, but PDL has a higher service level abstraction through the embedded cloud, as presented in Chapter 5.

4. REVIEW ON INFRASTRUCTURE ABSTRACTIONS FOR HETEROGENEOUS SENSOR NETWORKS

This chapter reviews infrastructure abstractions for heterogeneous sensor networks. The amount of existing research on infrastructure abstractions is overwhelming. Instead of covering all of the proposals, this review presents the diversity of this research field. As a result, common features are summarized, categorization is presented, and open questions on infrastructure abstractions are discussed. Publications that take WSNs design characteristics into account were selected for this review.

4.1 Infrastructure Abstractions for End User Applications

A typical infrastructure abstraction unifies heterogeneous sensing data sources and separates end user application implementation from the utilized sensing technologies. In this context, heterogeneity means using several sensing technologies without compatible low-level communication methods for cooperation. End user applications typically require data from different sources such as WSNs, wired sensors, databases, and automation systems, since one WSN is not capable of delivering all of the required data.

The resource constraints of the WSNs do not directly influence the design of the infrastructure abstractions. A server computer or embedded PC is harnessed to execute the implementation. Therefore, existing Internet technologies, distributed computing, and databases are used in the designs. However, the infrastructure abstraction should realize the WSN characteristics to abstract dynamic WSN behavior, and avoid unnecessary resource consumption.

4.2 Common Features of Infrastructure Abstractions

The common features of infrastructure abstractions are summarized in Table 8. Similar work has been proposed in [29, 150, 222], but they concentrate on a specific field:

[150] discusses Service-oriented Architecture (SOA) approaches, [222] discusses Sensor Web approaches, and [29] discusses Open Geospatial Consortium (OGC) Sensor Web Enablement (SWE). Table 8 presents the requirements from all aspects of infrastructure abstractions.

The main feature is establishing technology interoperability, which requires three components. Data access is unified, service discovery is provided, and the knowledge of the data is unified with an ontology. In addition, meta-data is required to complete the information and service access. The data source may be selected according to its accuracy or the sensor data may be illustrated on geographical maps. Finally, the sensing data is processed to create new refined data for end user applications.

4.3 Infrastructure Abstraction Categories

This thesis presents three categories for infrastructure abstractions: homogenization interfaces, Sensor Webs, and Sensor Clouds. These categories have vague borders as most of the reviewed works contain overlapping features [29]. Therefore, all reviewed works could be positioned under the homogenization interface category, and the categories partly built on top of each other. Figure 16 illustrates the relations and main topics of these three categories.

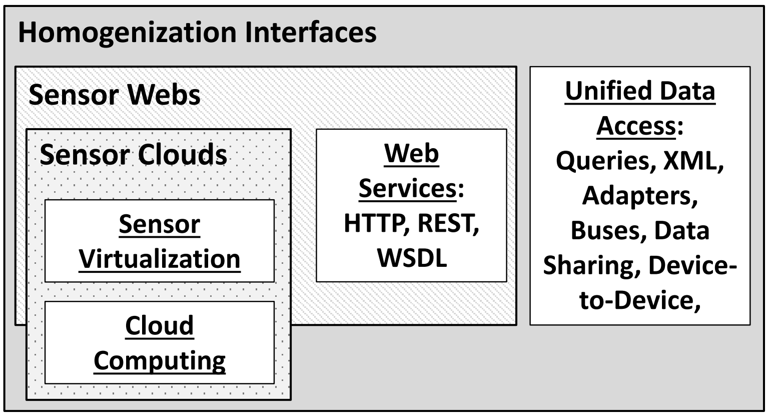

Homogenization Interfaces
Sensor Webs
Sensor Clouds
Sensor Virtualization
Cloud Computing
Web Services: HTTP, REST, WSDL
Unified Data Access: Queries, XML, Adapters, Buses, Data Sharing, Device-to-Device,

Fig. 16: Infrastructure abstraction categories and their relations.

Table 8: *Common features of infrastructure abstractions.*

Feature	Content	Purpose	Challenges	Example Proposals
Technology interoperability	Unified data access, service discovery, ontology	Homogenize heterogeneous sensors and actuators.	Incompatible accuracies and access methods	Sensor Webs as OGC SWE [25]. Virtual sensors as in [7,130,137].
Unified data access	Data delivery: continuous stream of data, history queries, delivery on triggered event.	Provide unified access to the sensor data and actuators	Unification requires technology specific adapters	Streams: OGC SWE SOS [32]. Queries: Global Sensor Networks (GSN) [1]. Events: OGC SWE SAS [183]
Service Discovery	Request services globally, geographically, or according to requirements. Register, publish, and subscribe services.	Discover available homogenized measurement, control and processing services.	Appearing and disappearing devices make services dynamic and unpredictable.	Sensor capability queries in OGC SWE SPS [185]. Service registration in TinySOA [19]. Meta-data searching in GSN [1].
Ontology	A schema to present data formats and connections.	Defines machine readable knowledge to represent the data with the same units and meta-data.	Unlimited amount of measuring and controlling quantities that ontology should cover.	XML in OGC SWE [26, 46], Lamses [98], and SeNsIM [35]. SSN [208] OWL. Smart-M3 ontology API.
Meta-data	Additional information for the application.	Enable service discovery based on location, accuracy, sampling rate etc. features. Complement the measurement data with meta-data.	Unlimited possibilities for the required meta-data. Application specific requirements.	SensorMap [158], SSN [208], O&M [46], SensorML [26]. Virtual sensors contain meta-data in GSN [1].
Processing	Data aggregation, data fusion, event detection, event creation.	Refine and fuse measurement data for the end-user application.	Application requirements vary, different sensor sampling rates, distribution of processing is not used.	Combined processing elements [27]. Virtual sensors GSN [1]. OGC SWE SAS pattern matching [183], SensorML process chains [26]

4.3.1 Homogenization Interfaces

A homogenization interface has two basic features: 1) it homogenizes data, and 2) it homogenizes the device access of heterogeneous sensor networks. All related work in this chapter implements these two features. Therefore, the following section reviews and discusses homogenization only interfaces. The homogenization interfaces can be divided into four subcategories.

1. **Data Query Interfaces** provide homogenized query access to sensors and their data. They resemble WSN as database middleware from network abstractions. These interfaces typically combine XML and SQL for data access and allow the incorporation of meta-data and processing [1, 98].

2. **Adapter and Bus Interfaces** connect devices together with one homogeneous connection. An end user application can utilize the connected devices by joining the same bus using an adapter or an interface provided by the infrastructure abstraction [35, 102].

3. **Device-to-Device Interfaces** allow devices from different technologies to directly communicate through one common connection, such as the Internet [43, 91, 204]. These interfaces allow the device itself to access the data of other devices, in addition to serving data requests from end user applications.

4. **Data Sharing Points** allow devices to publish, subscribe, and query data arbitrarily from a specific communication point [74, 85]. These points resemble the shared memory proposals for WSN middleware. The data sharing point defines an interface that the joining devices implement.

Table 9 summarizes the related research on homogenization interfaces. The proposals are categorized according to the subcategories presented above.

Since Constrained Application Protocol (CoAP) [43, 91] homogenizes only device access, not the data, it does not fit directly with homogenization interfaces. However, CoAP is a potential protocol to be the de facto standard for device access in future infrastructure abstractions, since it is standardized by the Internet Engineering Task Force (IETF) [180] to provide Hypertext Transfer Protocol (HTTP) access over the Internet to resource constrained devices.

CoAP has been used in *Californium* [111] to move application logic from WSN nodes to the cloud, in *Actinium* [110] for direct access to WSN nodes from JavaScript, in

T-RES [8] to distribute processing over WSN, and by Leppanen et al. [122, 123] to distribute mobile agents over heterogeneous IoT devices.

With resource constrained WSNs, CoAP works on top of IPv6 over Low power Wireless Personal Area Networks (6LoWPAN) [92]. 6LoWPAN compresses Internet Protocol version 6 (IPv6) headers so that IPv6 addressing can be used with IEEE 802.15.4 networks [90]. Similarly, CoAP compresses HTTP headers. Together they compress headers of originally several hundred bytes into a lightweight format of under 100 bytes, which enables HTTP Representational State Transfer (REST) access to resource constrained WSNs with IPv6 addressing. HTTP REST is explained in Section 4.3.2.

Homogenization interfaces would benefit from de facto standards for sensor data access and format. Currently there are several promising specifications from such organizations as the Semantic Sensor Network (SSN) from [208] the World Wide Web Consortium (W3C), which is an Web Ontology Language (OWL) [206] data knowledge homogenization for sensor networks, and CoAP [180] from IETF, which homogenizes data access.

UVPN, T-RES, Smart-M3, and Gómez-Goiri and López-de-Ipiña proposal allowed devices to decide when and how to connect to other devices. In the rest of the presented related works, device-to-device connections are established by the applications on top of the infrastructure abstraction. If a device cannot connect to other devices itself, the in-network processing is dependent on the network abstraction and its transparent use in the infrastructure abstraction, but the transparency breaks the homogenization. For example, a virtual sensor in GSN [1] can have a TinyDB query that uses in-network processing, but there is no homogenization between different technologies. Creating virtual sensors for all possible in-network processing queries is not feasible, either. As a result, a homogenized processing method is needed that maps the processing tasks of the infrastructure to the available in-network processing.

Although homogenization interfaces typically have methods for further processing and refining the data, they do not utilize the processing capabilities of the abstracted devices. GSN [1], proposal of Bouillet et al. [27], and SeNsIM [35] provided transparency for in-network processing, but did not utilize it on the processing executed on top of their abstraction. WOAG [102] utilized sensors as data providers only, although WOAG itself can be distributed between different gateway devices. Currently, T-RES [8] is the only related work that has realized in-network processing and device-to-device communication in its infrastructure abstraction.

Table 9: *Summary of the related research on homogenization interfaces.*

	Related research	Description
Data Query Interfaces	GSN [1]	Describes virtual sensors with XML, which consist of several input streams. SQL queries are used to discover, get, and process the data. The virtual sensors are grouped in containers that can form a peer-to-peer network over the Internet.
	Lamses [98]	Adds context-aware processing to an XML and query interface, which extracts events from the sensor data. Queries are extracted from the XMLs and used to access the homogenized sensor networks.
Adapter and Bus Interfaces	SeNsIM [35]	Unifies data with adapters to an XML query interface. The querying is provided to all the devices and end-user applications that join the bus.
	WSN OpenAPI Gateway (WOAG) [102]	WOAG instances form a distributed network, where instances run on gateway devices of different execution capabilities. The instances exchange data in the WSN OpenAPI format [P5].
	Bouillet et al. [27]	Uses processing elements that are described as semantic graphs constructing from a combination of tuples and ontology. The processing elements can be connected to refine, and modify the data streams.
Device-to-Device Interfaces	Ubiquitous Virtual Private Network (UVPN) [204]	Forms a virtual private network between sensor networks. Adapters homogenize devices that connect to UVPN endpoints. The endpoints are connected with virtual switches that forward messages over different protocols.
	CoAP [43, 91]	A header compression protocol for HTTP REST access to resource constrained IoT devices.
	T-RES [8]	Distributes Python scripts over WSNs with CoAP. Uses in-network processing on WSNs instead of external logic.
Data Sharing Points	Smart-M3 [85]	Allows devices store, retrieve, and modify data from an information store through a broker. The data is stored as triples according to some known ontology. An ontology API provides methods to discover the format of the stored data.
	Gómez-Goiri and López-de-Ipiña [74]	Uses HTTP REST API to form tuple space for data sharing between different devices.

4.3.2 Sensor Webs

Sensor Webs homogenize sensor data using Web Services [18, 29] that implement machine-to-machine services using protocols related to the World Wide Web (WWW)

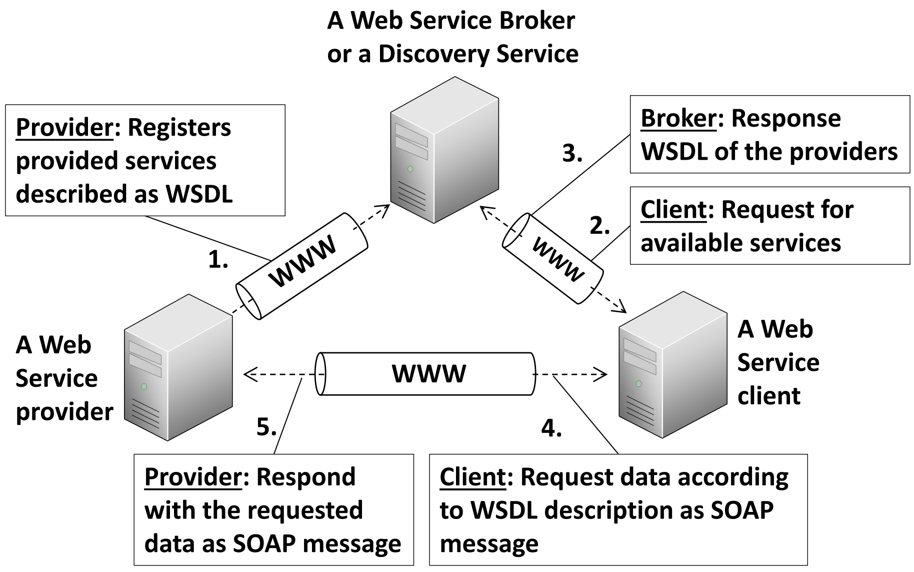

A Web Service Broker or a Discovery Service
Provider: Registers provided services described as WSDL
1.
WWW
3.
Broker: Response WSDL of the providers
2.
Client: Request for available services
WWW
A Web Service provider
WWW
A Web Service client
5.
4.
Provider: Respond with the requested data as SOAP message
Client: Request data according to WSDL description as SOAP message

Fig. 17: *A typical process of Web Services with WSDL and SOAP [209].*

[209]. Typically, a client connects to server resources via a Uniform Resource Identifier (URI) using HTTP messages. This approach is called REST architecture if the server-client communication is stateless [67].

Figure 17 presents a typical Web Service architecture built with Web Services Description Language (WSDL) and Simple Object Access Protocol (SOAP) XML messages [207, 209, 210]. Web Service interfaces are defined with WSDL XML. WSDL contains information about how the Web Service is accessed, and how the SOAP messages are constructed. The client discovers available services from the Web Service brokers as WSDL specifications. The client requests data using SOAP messages. The service provider responds with the requested data as SOAP messages. It should be noted that WSDL and SOAP are not bound to HTTP or REST architecture.

The term Sensor Web is ambiguous, since not all Sensor Web proposals utilize standard Web Service technologies. The term SOA is typically used with Sensor Webs as Web Service is one design pattern of SOA approaches. SOA sensor network abstractions have been surveyed in [29, 150, 222]. The term Web-of-Things is sometimes also used for Sensor Webs that use HTTP REST [29]. Due to their similarities, SOA and Web-of-Things are categorized under Web Services in this thesis.

Table 10 summarizes related research on Sensor Webs. OGC SWE is presented in detail in the following, since it is one of the most often referred to and studied Sensor Web proposals [25, 184] and represents a typical Sensor Web proposal.

Table 10: *Summary of the related research on Sensor Webs.*

Related research	Description
OGC SWE [25, 184]	Set of XML specifications for Sensor Web implementations.
Hourglass [182]	One of the first SOA proposals for connecting sensor networks and applications. Abstracts service providers behind XML described interconnected circuits that allows intermittent connections between service producers and consumers.
SenseWeb [76] and SensorMap [158]	SenseWeb uses adapting gateways to form a unified Web Service API. SensorMap has Web Service tools to visualize data from SenseWeb on georaphical maps.
HYDRA [63, 64, 95]	Tunnels SOAP messages through a resource aware middleware to achieve device-to-device interoperability. It is unclear, if HYDRA is usable with resource constrained WSNs.
SOCRADES [188]	Extends an enterprise Web Service architecture with a service proxy, which provides IoT devices as virtual devices.
Gomez and Laube [75]	Similar to SOCRADES. Provides context aware processing for business applications.
TinySOA by Aviléss-López and García-Macías [19]	Integrates WSN nodes to a Web Service interface with a SOA middleware and a gateway. The gateway registers WSN node services to a registry at a server. Not affiliated with the WSN network abstraction TinySOA [173, 174].
Amudson et al. [10]	WSN nodes implement a middleware that executes service graphs describing application functionality. WSNs are connected through gateways to the Internet as WSDL described Web Services.
Young-Jun Jeon et al. [96]	An end-to-end WSN to Web Service architecture that translates HTTP messages directly to WSNs, and allows access and processing data from a web browser.
Leppänen et al. [122, 123]	A CoAP based mobile agent proposal that distributes processing over heterogeneous IoTs. The mobile agents contain task code, information of the required local and remote resources, and the state of the agent. The state is stored over migration. The resources are accessed with Web Services.

OGC SWE currently consists of six XML interface specification for Sensor Webs. The specifications are Sensor Model Language (SensorML) [26], Observations and Measurements (O&M) [46], Sensor Alert Service (SAS) [183], Sensor Observation Service (SOS) [32], Sensor Planning Service (SPS) [185], and Web Notification Service (WNS) [186]. Table 11 summarizes these specifications and their dependencies, and Listing 4.1 gives an example of O&M.

Table 11: *Summary of OGC SWE specifications.*

Specification	Content	Dependencies
SensorML [25,26,184]	Describes processes and process chains. Each process has inputs, outputs, and parameters. Process without an input is a data source, e.g. a sensor. Geolocation and meta-data are included to process descriptions.	
O&M [46]	Defines observation data format. Combines meta-data, result, location, and observation time as presented in Listing 4.1.	
SOS [32,184]	Queries observations with various parameters, e.g. location.	Observations are delivered with O&M notation
SPS [184,185]	Configures tasks (e.g. sampling rate of a sensor) on the available sensors. Discovers sensor capabilities. Controls actuators.	Data access through SOS or other methods.
SAS [183,184]	Delivers and creates alert notifications. Creation with pattern matches. Sensors publish alerts to SAS. Consumers subscribe alerts from SAS.	WNS delivers notifications to subscribers
WNS [184,186]	Delivers notifications to the subscribers.	

The following specifications are related to OGC SWE, but are not part of the OGC SWE standards. Transducer Markup Language (TML) [82] describes a hardware model for sensors and actuators. TML was part of the original OGC SWE, but it is now a retired specification [184]. Sensor Instance Registry (SIR) [100] is a replacing interface for TML. It handles the meta-data of the sensors. SIR and Sensor Observation Registry (SOR) [99] provide discovery service for sensors and observations [29]. Sensor Event Service (SES) [62] is a replacement proposal for SAS. SIR and SOR are planned to be part of the OGC SWE standard. PUCK [162] is an OGC standard that can be used in conjunction with SWE. PUCK defines a protocol for connecting RS232 and Ethernet sensors and actuators.

OGC SWE specifications have been studied in the literature, including performance analysis [194,198], evaluation of usage with testing scenarios [142,169], evaluation with implementation for TinyDB and Mica Motes [39], evaluation with implementation for mobile sensors [149], and a survey of existing deployments that implement OGC SWE interfaces [195].

```
<om:OM_Observation>
  <gml:description>Observation: Room temperature</gml:
      description>
  <gml:name>Observation</gml:name>
  <om:phenomenonTime>
    <gml:TimeInstant
     gml:id="ot1">
       <gml:timePosition>1984-11-08T2:16:00.00</gml:timePosition>
    </gml:TimeInstant>
  </om:phenomenonTime>
  <om:result
     xsi:type="gml:MeasureType"
    uom="Cel">23.0</om:result>
</om:OM_Observation>
```

Listing 4.1: *A simplified O&M example of temperature measurement. Several required tags have been deprecated for the clarity of presentation.*

OGC SWE is a complete set of specifications for abstracting sensor networks as Web Services. Many implementations do not include teh entire set or the implementation is invalid. The core operations of SOS were typically implemented and 29% of the found instance files were invalid according to [195]. The invalidity can cause interface utilization problems. However, OGC SWE does not require all of the interface operations to be implemented, only the core operations.

OGC SWE is too complex to be implemented on a resource constrained WSN [130]. It is suitable for resource-rich nodes, gateways, or servers. Also, OGC SWE does not propose methods for distributing processing to sensor networks, but only utilizes sensors as data providers.

Sensor Webs integrate sensor networks into Internet applications. However, WSN nodes are not directly accessible with the HTTP REST architecture of Web Services, unless CoAP or similar compression protocol is used. As the CoAP builds on top of 6LoWPAN, CoAP requires IPv6 to become more common before becoming a de facto access method.

Currently, most of the Sensor Webs do not utilize or implement in-network processing on WSNs. CoAP allows processing to be extended to WSNs in REST architecture, as shown by T-RES and Leppänen et al. However, two problems exist. First, Web Services typically use XML, which contains too much data for transfer over a WSN. The data must be reformatted and compressed. Second, WSNs require new methods for transferring processing from the high-level descriptions of the Web Services to

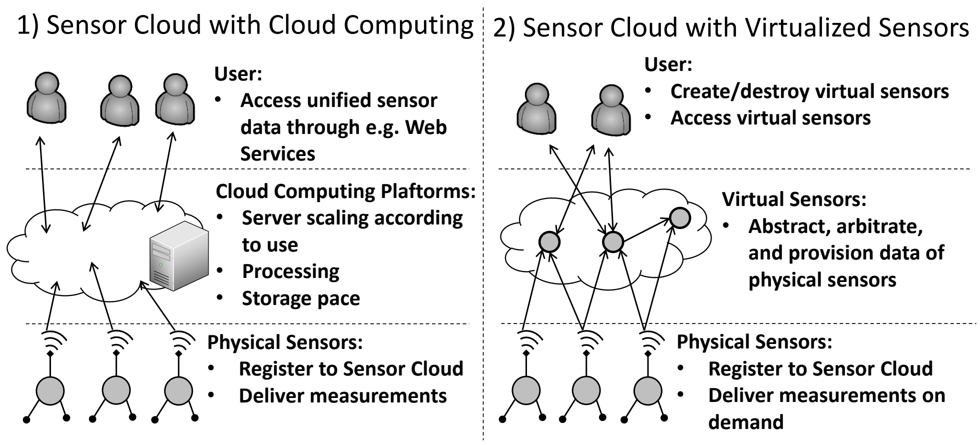

1) Sensor Cloud with Cloud Computing
User:
• Access unified sensor data through e.g. Web Services
Cloud Computing Plaftorms:
• Server scaling according to use
• Processing
• Storage pace
Physical Sensors:
• Register to Sensor Cloud
• Deliver measurements
2) Sensor Cloud with Virtualized Sensors
User:
• Create/destroy virtual sensors
• Access virtual sensors
Virtual Sensors:
• Abstract, arbitrate, and provision data of physical sensors
Physical Sensors:
• Register to Sensor Cloud
• Deliver measurements on demand

Fig. 18: *Sensor Cloud approaches.*

the WSN nodes.

4.3.3 Sensor Clouds

Sensor Clouds combine the cloud computing paradigm of *Infrastructure, Platform, or Software as a Service* with sensor networks [7, 130]. The term Sensor Cloud is used with two definitions: 1) virtualized server resources of cloud computing platforms are used with sensor networks; 2) sensors are virtualized in a similar manner as how server hardware and software are virtualized in Infrastructure or Platform as a Service cloud computing [7, 130, 137, 224, 225]. These two methods are depicted in Figure 18 and summarized in the following paragraphs.

A cloud computing platform scales the processing, bandwidth, and storage resources of a server infrastructure according to the demand of the application, with minimum effort or even automatically [7, 130]. Due to the scalability, the implementation of infrastructure abstraction can start on a small server with a small cost, and scale up if the demand for resources increases. This is beneficial when there is no definite forecast for the amount of end users or connected sensor networks. Sensor Clouds typically use Web Services [7, 130], as the cloud computing platforms are reached through the Internet.

The following proposals integrate cloud computing platforms with WSNs. A network of human health and activity recognition for care services is presented in [105, 120]. It integrates WSNs, context aware cameras, and a cloud computing platform into one application. IoTCloud middleware provides Web Service for Internet connected IoT devices to connect to the cloud [70]. A sensor cloud is presented in [121] that connects sensor networks through Amazon Web Services onto the Amazon EC2 cloud computing platform.

A cloud computing platform separates physical server HW from virtual HW with provided virtualization. Users can create hundreds of small servers that can be located anywhere on a large distributed server farm. A Sensor Cloud with virtualized sensors uses the same concept. A virtual sensor requested by the user is separated from the actual physical sensors [224]. The virtual sensor handles the communication and sensor data acquisition from the physical sensors. Virtual sensors can be further interconnected to form hierarchical processing. When the data is no longer needed, the virtual sensor can be destroyed.

The actual sensor data and data access are automatically provisioned in the virtual sensors [7]. Provisioning means ensuring the reasonable use of the physical sensors. For example, if two virtual sensors require the same data but with a different sampling rate, only one best fit sampling rate is selected. Provisioning also ensures that if two or more sensors are available that produce the required measurement, the sensor usage is distributed. If one of the sensors fails, its load is balanced to other sensors, and the failures are hidden from the user.

One of the first sensor virtualizations and automated provisioning system was presented as an architecture model in [224, 225]. The model abstracts physical sensors as virtual sensors and virtual sensor groups. The end-users have access to the sensor data through a web portal. The virtual sensors and groups are provisioned automatically according to the user needs.

WSN-SOrA [12] provides WSN as Network as a Service (NaaS). It is a SOA that provisions abstracted WSNs to multiple concurrent users. The service provisioning is done using XML. The prototype is implemented for TelosB resource constrained WSN platforms.

Publish/subscribe Software as a Service (SaaS) architecture was presented in [81]. It consists of application specific virtualized services that utilize the connected WSNs through publish/subscribe brokers. Provisioning and access policy are included in the design. A similar design was presented in [6] for IoTs. The measurement data is expressed with SensorML and access to services is authorized using access policies.

Sensor Clouds are either Sensor Webs with scalable cloud computing server or Sensor Webs extended with virtual sensors for sensor provisioning and accounting. Thus, the same issues concerning Sensor Webs in Section 4.3.2 concern Sensor Clouds.

The provisioning of physical sensors hides disappearing nodes and allows resource aware data acquisition without WSN-specific knowledge from the user. This separates Sensor Clouds from Sensor Webs. In-network processing should be added to these virtual sensors to make this approach more efficient for WSNs.

4.4 Research Questions in Infrastructure Abstractions

Seven open research questions for infrastructure abstractions are presented in [P4] about service discovery, security, privacy, QoS, de facto ontology, accounting, and performance metrics.

For this thesis, the main research question is that current infrastructure abstractions neglect the processing capabilities of the sensor devices. Most of the infrastructure abstractions utilize sensor devices only as data providers [8, 130, 166, 197, 204]. By allowing device-to-device communication [166,204] and in-network processing [190, 204], the heterogeneous sensor devices can form intelligent ubiquitous applications in co-operation to overcome resource constraints [37].

Other question for this thesis concern the lack of a de facto ontology for homogenizing data access. The wide application field results in a situation where the ontology should cover a wide range of possible measurements and application requirements. On the other hand, the ontology should contain only a reasonable and manageable amount of data structures, data units, and meta-data. Smart-M3 uses ontology APIs to address the different ontologies, but this does not reduce the tailoring effort in the way that a de facto ontology would reduce it.

As an update to the open questions in [P4], the lack of performance metrics has been partly addressed in [198] and [194]. The response times of different filtering methods of OGC SWE SOS were tested in [198]. In [194], sizes and processing times of OGC SWE were studied as plain XML, compressed XML, and JavaScript Object Notation (JSON). However, there is no comparability between the results, and further studies are needed to establish de facto metrics for comparing infrastructure abstractions.

5. RESULTS ON INFRASTRUCTURE ABSTRACTIONS

This chapter summaries the results of this thesis for infrastructure abstractions. This thesis makes three main contributions:

- The survey in [P4], freshened by Chapter 4, reviews, categorizes, and summarizes infrastructure abstractions.
- In publication [P5], a unified information model is presented that unifies data and data access for the WSN applications.
- An embedded cloud is proposed in [P6] that allows the distribution of processing over heterogeneous technologies and extends the resources of resource constrained WSNs.

5.1 An Information Model for Unified Measurement and Actuator Access

WSN OpenAPI is an infrastructure abstraction that describes data formats and interfaces for sensor and actuator network applications [P5]. WSN OpenAPI homogenizes sensor network data producers through adapters. It consists of six specifications:

1. Authentication and Capability Format (ACF) describes messages for authenticating connections to WSN OpenAPI gateway.
2. Network Management Format (NMF) describes message formats for querying network status, and configuring network data collection and alerts.
3. Meta-Data Format (MEDF) describes message formats for querying and inserting node capabilities and their sensors in SensorML format.
4. Sensor Information Data Format (SIDF) describes the format for measurement data.

5. Sensor Archive Data Format (SADF) describes a request-response interface for accessing archived data.

6. Node Actuator and Sensor Control (NASC) provides an interface for controlling actuators and sensors.

The SIDF, SADF, and NASC interfaces follow the hierarchy presented in Figure 19. All of the message formats and interfaces are described using XML and Comma Separated Values (CSV). The CSV variant was developed as a compressed format for the resource constrained environments, e.g. sending data over low bandwidth connections such as WSNs or General Packet Radio Service (GPRS).

WSN OpenAPI is a light-weight measurement and actuator information model, when compared to OGC SWE and similar proposals that target covering excessive amount of situations. For example, the entire WSN OpenAPI technical specification is 119 pages [196], where as the original OGC SWE specifications are 1077 pages in total (including the retired TML) [26, 32, 46, 82, 183–186].

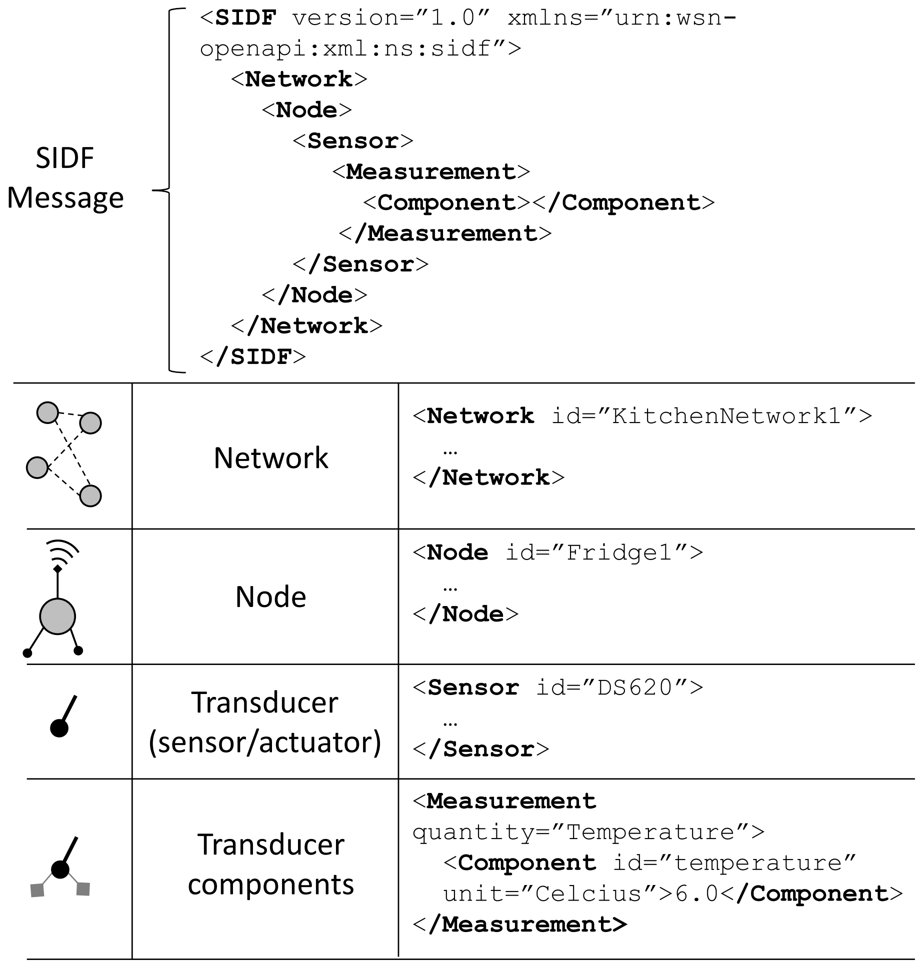
WSN OpenAPI has been used in WOAG to describe data formats and as an access method [102]. The distributed WOAG instances communicate using WSN OpenAPI as well.

Since there is no de facto ontology, WSN OpenAPI has been used as an information model by using its network-node-measurement hierarchy [P4]. In this method, all data sources are treated as measurement data. Three example use cases are presented in the following.

1. Integrating social media as part of a ubiquitous sensor network application has been envisioned [18]. As an example, a Facebook status update can be considered as a measurement that has a value (the status update), a unit (text or picture), and a time stamp. Also, a person is a node in a network of friends.

2. The Finnish Meteorological Institute (FMI) provides Hirlam weather forecasts as part of their open data service [68]. Where as real time weather observations are measurements, forecasts spread over the following 48 hours and future forecasts change every hour. Therefore, one set of forecasts is handled as one measurement of that specific time.

3. Data provided on the Internet may be needed in WSN applications. Similar to Facebook integration, a website is a network of web pages (nodes) and the contents of the web page can be measured at a specific time.



```
<SIDF version="1.0" xmlns="urn:wsn-
openapi:xml:ns:sidf">
  <Network>
    <Node>
      <Sensor>
        <Measurement>
          <Component></Component>
        </Measurement>
      </Sensor>
    </Node>
  </Network>
</SIDF>
```

	Network	`<Network id="KitchenNetwork1">`
		`  …`
		`</Network>`

	Node	`<Node id="Fridge1">`
		`  …`
		`</Node>`

	Transducer (sensor/actuator)	`<Sensor id="DS620">`
		`  …`
		`</Sensor>`

	Transducer components	`<Measurement`
		`quantity="Temperature">`
		`  <Component id="temperature"`
		`  unit="Celcius">6.0</Component>`
		`</Measurement>`

Fig. 19: *The hierarchy of a WSN OpenAPI SIDF message.*

5.2 Embedded Cloud

This thesis proposes an embedded cloud as a solution for the lack of in-network processing in infrastructure abstractions. The embedded cloud binds the node, the network, and the infrastructure abstractions in to one design that resembles SOA. The embedded cloud abstracts measuring, actuating, and processing as services. The embedded cloud shares all of the available services in a domain between the connected device from different technologies. Upon a connection, the device registers its services to the embedded cloud to share them with the other connected devices. The domain can be used to restrict visible services according to the physical location

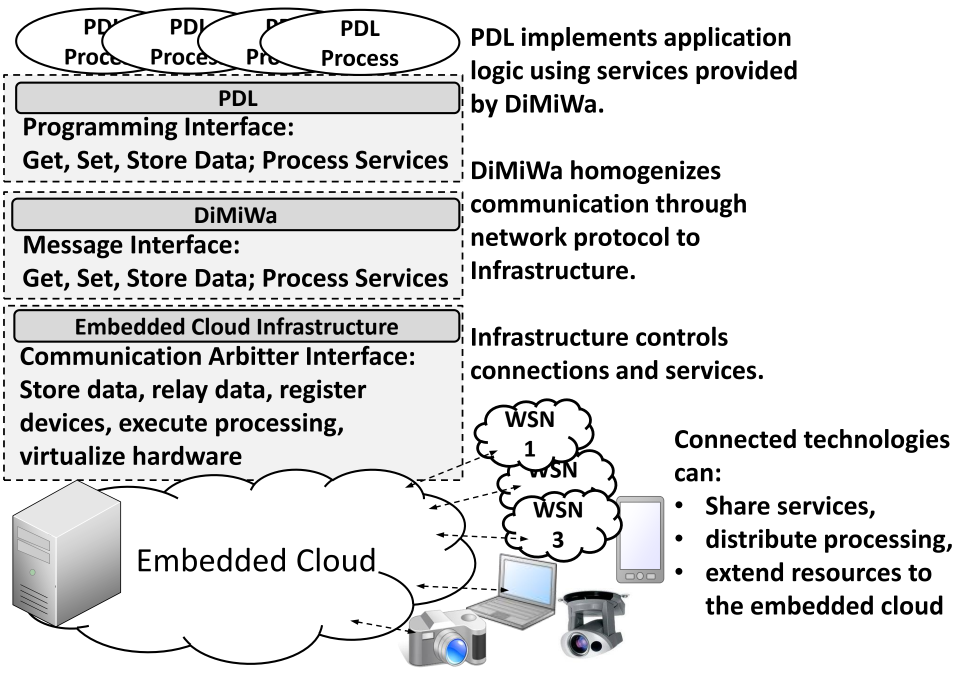

PDL Process
PDL implements application logic using services provided by DiMiWa.
PDL
Programming Interface:
Get, Set, Store Data; Process Services
DiMiWa homogenizes communication through network protocol to Infrastructure.
DiMiWa
Message Interface:
Get, Set, Store Data; Process Services
Embedded Cloud Infrastructure
Communication Arbitter Interface:
Store data, relay data, register devices, execute processing, virtualize hardware
Infrastructure controls connections and services.
WSN 1
WSN
WSN 3
Connected technologies can:
• Share services,
• distribute processing,
• extend resources to the embedded cloud
Embedded Cloud

Fig. 20: *The design of the embedded cloud.*

of the devices.

Since processing of the connected device is available as a service, the embedded cloud applications can utilize the processing power from any of the connected devices. As the in-network processing of a WSN is one service, the embedded cloud applications can utilize these in-network processing capabilities. Furthermore, the embedded cloud provides additional processing power and storage for the connected devices in a similar way to how the cloud computing platforms provide for traditional server software. These are the main distinguishing features when compared, for example, to the TinySOA of Aviléss-López and García-Macías [19].

The embedded cloud consists of three components: an infrastructure, DiMiWa, and PDL. These components and their relations are depicted in Figure 20 and explained in the following sections.

5.2.1 Embedded Cloud Infrastructure

The embedded cloud infrastructure works as a communication arbiter, a data storage service, and a processing service for the connected devices. For example, a resource constrained WSN node can store an image of a web camera to the embedded cloud and ask the embedded cloud to process it using a pattern recognition service. Then, the embedded cloud will deliver only the processing result back to the node. As a result, the WSN node can access data processing that would be too resource-consuming or even impossible on a resource constrained WSN.

Using DiMiWa, the devices register themselves to the embedded cloud infrastructure and available services are exchanged in the registration handshake. When the connected devices access the remote services, the infrastructure arbitrates the required messages. Upon new device registration, the infrastructure updates the remote services on the connected devices.

DiMiWa and PDL implementations are required for a device to connect to the embedded cloud. The infrastructure can implement DiMiWa and PDL through tailored adapters for technologies that are unable to execute software.

The embedded cloud infrastructure works as an infrastructure abstraction for end user applications. An end user application can access the data in the embedded cloud through a database connection. Also, end user applications can inject new PDL processes into the embedded cloud to get more refined data, or to execute application logic in the cloud.

5.2.2 DiMiWa: A Distributed Middleware

DiMiWa implements the service and the communication abstraction in each technology. DiMiWa is distributed middleware for two reasons:

1. To connect a device to the embedded cloud, DiMiWa can be implemented on the device itself, on an intermediate device, or on the embedded cloud infrastructure as depicted in Figure 21.
2. DiMiWa can be implemented on top of the node abstraction, on top of the network abstraction, or can implement the network abstraction in a WSN as depicted in Figure 21.

DiMiWa describes messages for the access, control, and use of the services in the embedded cloud. Each service has a domain where it is available, and a class of SAMPLE, EVENT, or BLOB. BLOB services produce an amount of data which cannot fit in the minimum data packet size of DiMiWa. The application cannot access

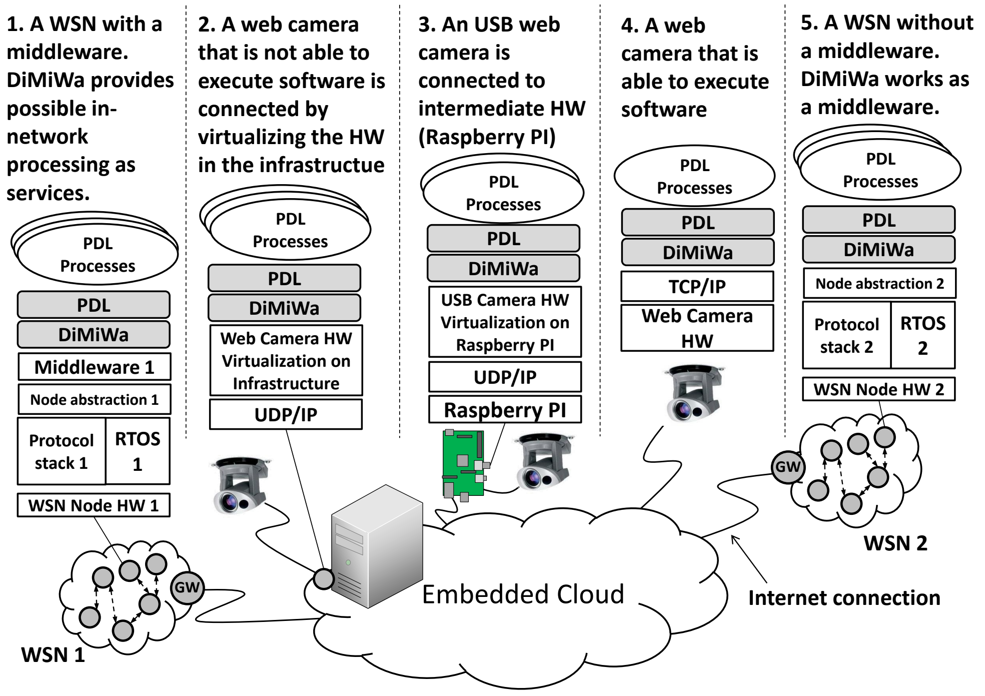

1. A WSN with a middleware. DiMiWa provides possible in-network processing as services.
2. A web camera that is not able to execute software is connected by virtualizing the HW in the infrastructue
3. An USB web camera is connected to intermediate HW (Raspberry PI)
4. A web camera that is able to execute software
5. A WSN without a middleware. DiMiWa works as a middleware.
PDL Processes
PDL
DiMiWa
Middleware 1
Node abstraction 1
Protocol stack 1
RTOS 1
WSN Node HW 1
Web Camera HW Virtualization on Infrastructure
UDP/IP
USB Camera HW Virtualization on Raspberry PI
Raspberry PI
TCP/IP
Web Camera HW
Node abstraction 2
Protocol stack 2
RTOS 2
WSN Node HW 2
GW
WSN 1
WSN 2
Embedded Cloud
Internet connection

Fig. 21: *The execution places of DiMiWa. DiMiWa can locate on the device, intermediate device, or virtualized hardware in the embedded cloud infrastructure.*

remote BLOB services directly, but the application can store them to the embedded cloud and use them as parameters for processing.

DiMiWa has a cache that stores the registered services and their newest data samples for SAMPLE and EVENT classes. A Time-To-Live (TTL) value is set on each service. The TTL is infinite for local and permanent remote services, such as data processing and wired sensors. For other remote services, TTL is a descending counter. If the remote service does not deliver a new data sample before the TTL reaches zero, the service is removed from the cache and the PDL processes which used the disappeared service are halted. This provides a service discovery method in the embedded cloud.

The requirements of DiMiWa for implementing technology are the ability to execute software, and communicate with the embedded cloud infrastructure. The communication requires sending and receiving a message size of 9 B at minimum that consists of a 1 B packet type identifier, a 4 B service identifier, and a minimum of a 4 B payload. If the technology cannot execute the software or send and receive messages, the DiMiWa implementation must be executed on the infrastructure or on the

Table 12: PDL actions, parameters and purpose.

Action	Parameters	Purpose
STORE	SERVICE	Stores value of the SERVICE to the cloud.
GET	SERVICE	Returns a value of the SERVICE. The value is stored to the internal ACCU.
SET	SERVICE	Sets the ACCU to the SERVICE.
SET_ACCU	DATA: a new internal value	Sets the immediate data to the ACCU.
TRIGGER	SERVICE	Blocks process until the SERVICE returns a triggered condition.
TIMER	DATA: wait time in seconds	Blocks process until the amount of seconds of the given data have passed.
TIMEWINDOW	DATA: time window in seconds	Restarts the process, if the time window is not cleared before expiration.
RESTART	-	Restarts the process.
JUMP	DATA: a jump offset	Take a jump of the offset length.
CONDITIONAL_JUMP	DATA: a jump offset	If the next ACTION will rise the TRUE flag, the process takes the jump. Offset is added after the conditional evaluation.
PROCESS_SERVICE	process SERVICE, source SERVICE	Give the source SERVICE as an input to the process SERVICE
PROCESS_ACCU	process SERVICE	Give the ACCU as an input to the process SERVICE
OPERATION	SERVICE, OPERATION	Execute “SERVICE OPERATION ACCU” equation. Set TRUE flag if needed.
OPERATION_IMMEDIATE	SERVICE, OPERATION, DATA	Execute “SERVICE OPERATION DATA” equation. Set TRUE flag if needed.
INC_ACCU	-	Increments ACCU with 1.
DEC_ACCU	-	Decrements ACCU with 1.

intermediate device, as shown in Figure 21.

5.2.3 PDL: Process Description Language

PDL allows device-independent application development on top of DiMiWa. The desired application functionality is described as a sequence of actions presented in Table 12. Actions have access to DiMiWa services and to an ACCU service that is a PDL process-specific intermediate register for arithmetic calculations. In addition to service access, a PDL process can set timers and triggers, manipulate the execution with jumps, and manipulate the ACCU. The PDL timers have a one-second resolution.

Listing 5.1 gives an example of using a pattern recognition service to detect humans from a picture. Line 2 sets a 30 s sampling frequency. Line 3 stores a picture. Line 4

```
1  uint8_t pdl_process[] = {
2    PDL_ACTION_TIMER, 0x00, 0x1E,
3    PDL_ACTION_STORE, DIMIWA_SERVICE_CAMERA_BYTES,
4    PDL_ACTION_PROCESS_SERVICE,
         DIMIWA_SERVICE_HUMAN_PATTERN_REGOCNITION_BYTES,
         DIMIWA_SERVICE_CAMERA_BYTES,
5    PDL_ACTION_TIMEWINDOW, 0x00, 0x1E,
6    PDL_ACTION_TRIGGER,
         DIMIWA_SERVICE_HUMAN_PATTERN_REGOCNITION_BYTES,
7    PDL_ACTION_SET_ACCU, 0x00,0x00,0x00,0x01,
8    PDL_ACTION_SET, DIMIWA_SERVICE_ALARMSOUND_ACTUATOR_BYTES,
9    PDL_ACTION_RESTART };
```

Listing 5.1: *A PDL process that detects a human from a picture and plays an alarm sound.*

asks the embedded cloud to process it with human pattern recognition. Line 5 sets a time window for the processing. Line 6 triggers a human detection message, unless the time window expires before. Lines 7 and 8 set the sound alarm actuator. More examples are given in [P6].

5.2.4 Proof of Concept Implementation

The proof of concept implementation concentrates on DiMiWa and PDL, as they are the components that need to operate on the resource constrained WSN nodes. A proof of concept implementation of the embedded cloud infrastructure was implemented with the C++ programming language using User Datagram Protocol (UDP) packets. The implementation of DiMiWa and PDL was made for the TUTWSN platform. In addition, both were tested on Raspberry Pi devices [171].

The DiMiWa implementation consumes 3222 B of program memory on a PIC18F8722 MCU with a Microchip MCC18 compiler [147]. Each cache entry utilizes 14 B of data memory, which results to over hundred possible simultaneous services [P6], if 50% of the available data memory is allocated for the cache. The implementation includes the messaging, local services, and the cache.

The proof of concept implementation of PDL is 354 lines of C code. It requires 1900 B of program memory on a PIC18F8722 when compiled with a Microchip MCC18 compiler [147]. Each PDL process requires 16 B of data memory. All PDL operations take under 1 ms to execute on a PIC18F8722 with a 4 MHz clock speed, which allows over 100 PDL processes to be executed, when PDL implementation is executed once every 100 ms [P6].

The proof of concept prototypes show that DiMiWa and PDL can be implemented and executed even on resource constrained WSN nodes, as the required memory resources occupy less than 5% of the available program memory. The number of possible simultaneous services and PDL processes exceeds the currently typical number of 2–6 application tasks [190]. Therefore, it is possible to implement and use the embedded cloud on resource constrained WSN nodes.

5.2.5 Comparison and Evaluation

As presented in Chapter 4, few infrastructure abstractions utilize in-network processing. Most notably, T-RES [8] and design of Leppänen et al. [122, 123] allow distributed processing over heterogeneous WSNs.

The Leppänen et al. design uses mobile agents and an REST interface for distribution. The devices can communicate and share resources through the interface and migration of the agents. The design has two major differences when compared with the embedded cloud:

1. The task code in the mobile agents is not homogenized. It can be arbitrary machine code, VM code, or interpret byte code that can be queried from a code repository. This requires the implementation of the task code for all possible technologies that are expected to migrate the mobile agents that use the task.
2. Combining arbitrary resources remotely is not possible with mobile agents, as they connect directly to the remote resources to get the data for the task code to process. The PDL process in the embedded cloud can connect any resource to a remote processing service. In theory, a mobile agent could mimic similar behavior through its migration. It is unclear from [122, 123] if this is possible.

The evaluation of Leppänen et al. concentrates on the implementation size, delays and message sizes. The design of Leppänen et al. consumes 28864 B of program memory and 3850 B of data memory on an 8-bit Atmel ATmega MCUs. These figures are larger than with the embedded cloud. The embedded cloud uses 1 s granularity in its functioning, as many low duty cycle WSNs are not able to deliver data with a lower delay. The total delay depends on the protocol stack utilized. Therefore, the point-to-point delay is not considered for the embedded cloud. The message sizes of Leppänen et al. vary between 8 B and 66 B toward the WSN nodes when using a

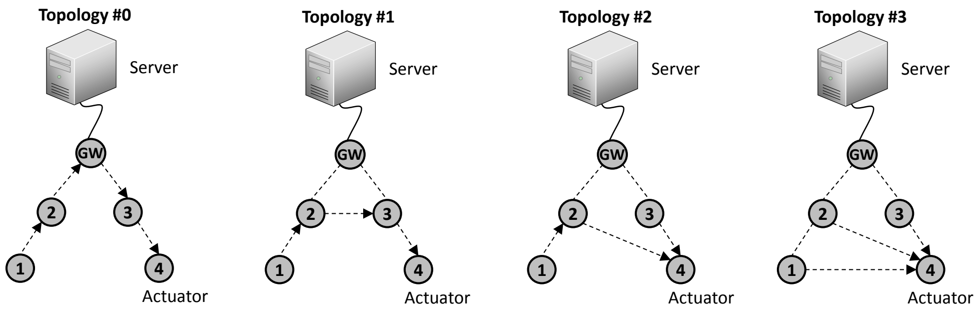


Fig. 22: *The topologies used to evaluate T-RES [8] and the embedded cloud.*

20 B task code. These are similar figures to the 9 B minimum message size of the embedded cloud.

T-RES is evaluated using simulations of four different topologies with applications that control actuators according to sensor values [8]. Figure 22 presents these topologies. Unfortunately, direct comparison is impossible, since [8] did not provide enough detail of the T-RES simulations to allow the embedded cloud to be fitted exactly to the same simulations. However, the same application and topologies can be used to compare the embedded cloud with the in-network processing to a server only logic similar to the T-RES evaluation.

The following assumptions were used for the embedded cloud analysis that was created using Excel: 1) there is no packet loss, 2) the network is only executing one PDL process on Node 4 that takes the average of the temperature from all four nodes including itself and adjusts the actuator accordingly. 3) the temperature is measured every 5 minutes, 4) each embedded cloud packet is 9 B except the PDL delivery packet, which is 9 B + PDL Process size of 62 B (such a PDL process is given in [P6]). Thus, the packet headers of the protocol stack are not calculated, 5) protocol stack-related handshaking and communication packets are not considered in this simulation, 6) at the time of 0, the embedded cloud has generated 340 B of initialization data traffic (registration, service pushing, and service subscription) whereas the server-only logic is assumed to generate 0 B.

Figure 23 presents the worst case scenario, where the in-network processing requires packet routing through the gateway and server. The embedded cloud results in slightly more traffic due to the initial 340 B overhead. The results have a similar form to T-RES.

Figure 24 presents the best case scenario, where Nodes 1, 2, and 3 can send temperature packets directly to Node 4. The in-network processing reduces data traffic signif-

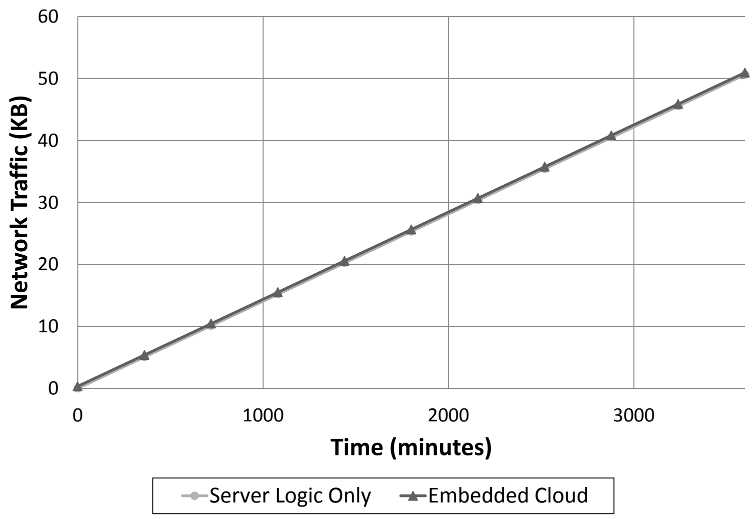


Fig. 23: *An analysis of the cumulative network traffic with Topology #0 for the heating controller application. This is the worst case scenario for the embedded cloud.*

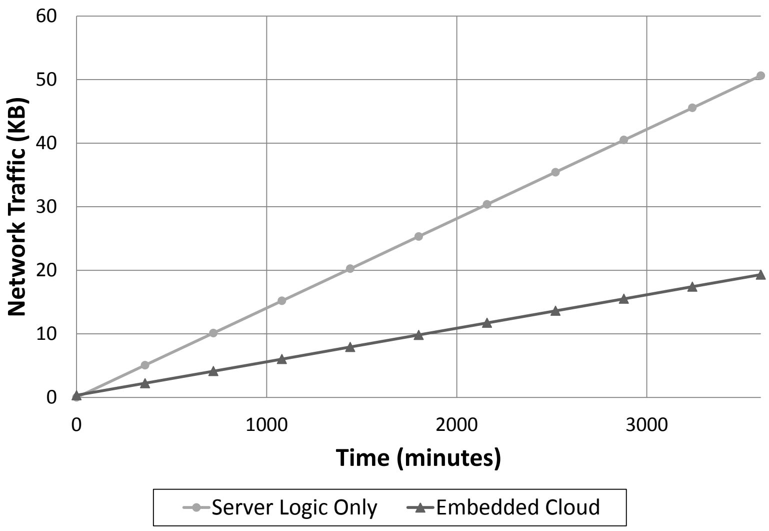


Fig. 24: *An analysis of the cumulative network traffic with Topology #3 for the heating controller application. This is the best case scenario for the embedded cloud.*

icantly, as expected from the related work on in-network processing [8,112,134,170].

Figure 25 presents network traffic at the 3600 minute mark for all of the topologies. The results are similar to T-RES and show the benefits of in-network processing when the topology is suitable. Unlike T-RES, the embedded cloud can execute a PDL

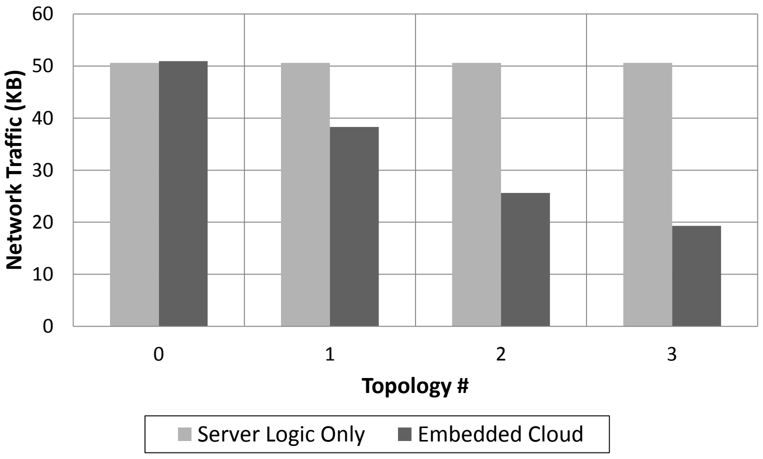


Fig. 25: *An analysis of the cumulative network traffic with all the topologies at the 3600 minute mark for the heating controller application.*

process on the embedded cloud infrastructure to ensure the best performance, if the topology is not suitable for efficient in-network processing. Furthermore, the T-RES implementation is heavier due to the python interpreter, which consumes 8 KB of data memory and 45 KB of program memory.

The energy consumption of the presented topologies and network traffic depends on the HW, WSN software and network protocols used. The overhead of running PDL processes is the main factor in the energy consumption in the embedded cloud. The actions of PDL processes are executed every 100 ms [P6]. All PDL actions take less than 1 ms on a PIC18F8722, which consumes 1 mA with 2.0 V supply voltage at a 4 MHz clock speed. Thus, one action of a PDL process takes the maximum of 0.76 μJ of energy. This results in ca. 630 J of energy consumed in a year, which is 3.15% of the 20000 J energy budget.

In practice, PDL processes wait for a timer, a trigger, or a get action most of the time. These actions take less than 200 μs of execution time [P6]. Thus, the best case energy consumption of one PDL process would be 126 J per year, which is 0.63% of the 20000 J energy budget. Figure 26 presents the energy consumption of 1 to 50 PDL processes in a year in the worst and best case. The PDL execution can be configured to execute PDL processes with round robin scheduling. Then, each 100 ms iteration executes only one PDL process action. This would limit the energy consumption of PDL execution to between 126 J to 630 J a year, with the drawback of a slower reaction to events.

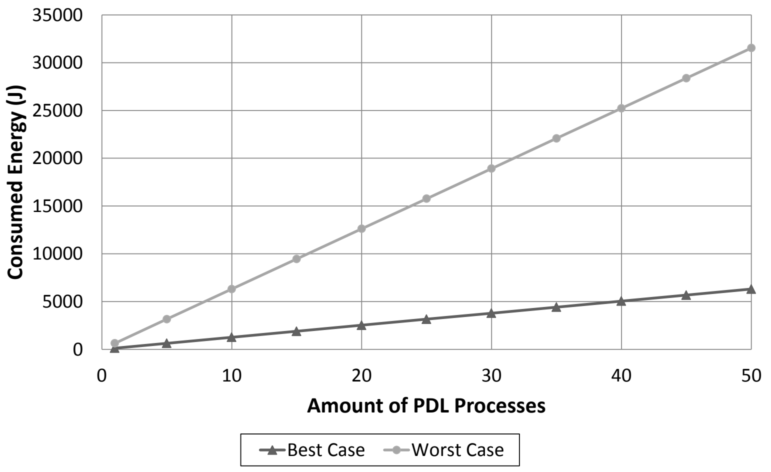


Fig. 26: *The energy consumption of PDL processes in a year for Microchip PIC18F8722.*

5.2.6 Future Work and Discussion

The presented embedded cloud solves distribution and shared service access between heterogeneous sensor networks. There are still four major open tasks for future work:

1. The design would benefit from an intelligent PDL process broker that decides where the execution of the PDL process is most efficient. This would require meta-data that models the execution and communication parameters of the devices and technologies.

2. The broker could automatically split PDL processes into subprocesses, if the subprocess could improve efficiency in the case that the technology is not capable of executing the entire PDL process. Currently, the PDL processes are executed where the required services are available.

3. An intelligent DiMiWa service broker could further enhance the efficiency by pushing only the required remote services to the devices. Currently, all services in the same domain are pushed to the devices.

4. Currently, the embedded cloud tracks an IP address and an ID for each device. The IDs need to be adapted when the technology is connected to the embedded cloud. This adapting could be avoided if the devices had a uniform addressing method, such as IPv6 addressing through 6LoWPAN. In addition, using CoAP would further reduce the tailoring by uniforming the message headers that encapsulate the embedded cloud messages.

Table 13 summarizes the research on the embedded cloud. Although the embedded cloud has future work to be done, it already allows heterogeneous devices from resource constrained WSNs to high computing power servers to share data, processing, and resources, which is not presented in the related works. PDL processes provide a homogeneous method for utilizing the abstracted services in the embedded cloud, which makes it possible to use in-network processing. As the foremost contribution, the embedded cloud shows that the processing and execution capabilities of resource constrained WSNs can be efficiently harnessed with an infrastructure abstraction.

Table 13: *Benefits, drawbacks, and future work of the embedded cloud.*

Feature	Benefits	Drawbacks	Future work
PDL	Low overheads with dynamic loading. Sense and react approach. Hardware independent.	Not suitable for low delay applications. Only 32 bit integer manipulation.	Create process generation from graphical data flow and automation tools.
DiMiWa	Service abstraction. Small packet overhead. Allows utilization of existing in-network processing as a service.	Services need a predetermined ID and homogenized data values.	Create an intelligent service broker.
Distributed processing	Application logic can be executed where best suited. Reduces data traffic.	Adds an execution overhead.	Create an intelligent PDL process and DiMiWa service broker for automated distribution.
Resource sharing	Allow resource constrained WSNs to execute applications that are not otherwise possible.	Increase in overheads when distributing data.	Increase intelligence with a PDL process and DiMiWa service broker.
Embedded cloud	An infrastructure abstraction that allows resource sharing and distributed processing even on resource constrained WSNs.	Requires adaptation from the technology. Domain and addressing are not automated.	Create intelligent brokers for addressing, services, and PDL processes. Distribute the embedded cloud infrastructure.

6. ANALYSIS OF WSN FIELD EXPERIMENTS AS A RESEARCH METHOD

This chapter analyzes the lessons of the WSN field experimenting for WSN application development. First, the related WSN field experiments are studied. Second, the contribution of this thesis to the lessons is presented. Finally, the lessons are compared with the abstraction results of this thesis.

6.1 WSN Field Experiments

Field experiments study WSNs in real and often harsh environments. They are utilized to study environmental effects on the functioning of HW, protocol, and application implementations.

Field experiment studies have been reported for varying use cases: industrial monitoring [113], habitat monitoring [138, 193], smart office space [119], macro climate monitoring [201], environment monitoring [44], hospital personnel and asset tracking [103], structural monitoring [163], and volcano activity monitoring [215], to present a few examples.

The related work states that field experimenting with a WSN can reveal unforeseen problems [21,44,87,118,168] and require costly iterations to designs and implementations. By analyzing the experiments for lessons, problems in future WSN development can be avoided.

Most of the field experiment publications give experiences of the suitability of WSN for the particular application use case. Four related publications analyze experiments to conclude lessons that are summarized in Table 14.

In [118], a monitoring application for a potato field was implemented. The implementation consisted of Atmel ATMega 128L based nodes with modified TinyOS, T-MAC, MintRoute, Deluge, a gateway, and the required application software. The deployment encountered several problems, such as hardware failures, casing problems, software errors, and overlooked software testing. Five lessons were listed from the experiences. Four of the lessons pointed out the complex software and distributed functioning, which make testing problematic when combined with resource constraints and limited connectivity. The fifth discussed assuming the worst for software development and environmental conditions.

Seven different deployments were presented in [21]. The deployments used TI MSP-430 based nodes with TinyOS, SensorScope communication stack, and application software. The lessons consist of software complexities, embedded software testing, remote controlling, remote monitoring, scientific publishing, research partnering, simulating, node casings, and data utilization to name a few. The major lessons were on using an incremental iterative development process, avoiding unnecessary complexities, and avoiding assumptions about data correctness and environmental conditions.

Corke et al. [44] presented nine field experiments with iterative WSN development of Atmel ATMega 128 based nodes. The lessons covered software complexities, difficult testing and debugging of remote deployments, enclosing problems, unexpected maintenance tasks, and unexpected environmental problems. In their iterative process, they found that each different environment set its own challenges: the RF reception changes when sugar canes grow, leaves may drop on solar panels, and water reflects RF differently than the ground. Following the software complexity lessons, they designed a tailored WSN OS during the iterations that had a more convenient programming approach for developers than TinyOS. Furthermore, the debugging and remote tools were improved to increase the productivity of the development. The lessons concluded with a description of end-to-end WSN that was thoroughly tested and where the application development was moved away from node-level programming with a remote procedure call system.

Hu et al. [87] covered a sugar cane field experiment by Corke et al. [44] in detail. They highlighted problems with software complexities, backbone network connections, unexpected environmental issues, and a requirement for watchdogs in remote experiments.

The reported deployment problems have been used as reasoning for in-deployment testbeds [23], since laboratory test benches, such as Motelab [216] and TOSSIM [126], are not sufficient to expose the problems that real harsh working environments can cause. In-deployment testing and debugging methods were proposed in [34, 60, 175, 217]. These methods allow debugging of the distributed WSN during or after deployment. The debug information is either delivered to the developer in real-time or logged on the nodes for later examination. However, these proposals do not remove

Table 14: *The lessons described by the related research. X denotes that the publication discusses the lesson.*

Lesson	[118]	[21]	[44]	[87]
Challenges of testing the distributed software	X	X	X	X
Complex software on resource constrained nodes	X	X	X	X
Inadequate node enclosures for harsh environment	X	X	X	-
Watchdogs required on nodes and on servers	-	X	X	X
Deployment monitoring, controlling and tracing required	X	X	X	-
Non-existing or unreliable backbone network connections	-	X	-	X
Tailored installation and deployment tools required	-	X	X	-
Expertise of partners and application field experts required	-	X	X	-

or solve the software complexities, which abstractions ought to solve.

6.2 Results of Analyzed Field Experiments

Publications [P2], [P3], and [P5] used WSN field experimenting as a research method.

- Analysis of 11 field experiments were presented in [P3]. Table 15 summarizes these experiments.
- PIDP was tested with a campus-wide TUTWSN deployment that consisted of 178 nodes in [P2].
- The greenhouse monitoring deployment was a use case for evaluating WSN OpenAPI in [P5].

This thesis presents six lessons from the WSN field experiments that were conducted with 1206 deployed resource constrained WSN nodes [P3]. Table 16 summarizes these lessons, which resulted in the end-to-end WSN architecture of TUTWSN. These lessons contribute to and reinforce the reviewed lessons of the related research.

The reviewed and contributed lessons provide reasons for the research on the WSN abstractions. Software complexities were mentioned in all of the related lessons. Removing software complexities is the main feature of the abstractions. Also, the

Table 15: *Summary of the field experiments studied in this thesis [P3]. LL TUTWSN stands for the low latency version of TUTWSN.*

Pilot Study	Nodes	Duration	Technology
Sewer water level monitoring	25	2009-2011	TUTWSN 433 MHz
Chemical factory monitoring	62	2009 (a year)	TUTWSN 2.4 GHz
Green house temperature and humidity leveling	30	2009 (a month)	TUTWSN 2.4 GHz
Campus network for teaching	340	2008-	TUTWSN 2.4 GHz
Home monitoring in several homes	180	2007-2011	TUTWSN 2.4 GHz
Transportation cargo monitoring	10	2009 (a year)	TUTWSN 2.4 GHz
Building monitoring	377	2008-2010	TUTWSN 2.4 GHz
Environment monitoring	60	2005-2011	TUTWSN 433 MHz
Environment monitoring, ground frost and snow depth	30	2007 (a year)	TUTWSN 433 MHz
Cattle living conditions in barn	30	2009-2011	TUTWSN 2.4 GHz
Hospital personnel safety	62	2009-	LL TUTWSN 2.4 GHz
Total:	1206		

field experiments give the practical problems that the abstraction designs need to realize.

The problems of complex software development with WSN nodes show that the node and network abstractions used have not been sufficient. For example, Deluge failed in [118] as it used the underlying network protocol stack to function. As the stack had unforeseen problems when deployed, the nodes required reprogramming through physical access. Also, the number of nodes required automation in the configuration and software programming.

The field experiments show the importance of not assuming the existence of data sources in the design of the infrastructure abstraction. All of the WSN field experiments encountered disappearing devices, or disappearing services, due to the dynamic and unpredictable environment.

Thoroughly tested end-to-end architecture and automated tools ensure the success of research [44]. The end-to-end architecture of Corke et al. [44] contained all of the abstraction levels presented in this thesis and used high-level description of the applications. This suggests that for WSN application development, satisfactory abstractions should realize all of the abstraction levels and be thoroughly tested, and the application development should take place at as a high level as possible.

The results of this thesis address the presented issues as follows. The OTAP method works autonomously and contains a fallback method that works separately to the protocol stack. The embedded cloud provides an end-to-end architecture that reduces

Table 16: Summary of the lessons [P3].

Challenge	Lessons learned
Deployments follow the same process, even if the application is different.	A systematic process should be used for all deployments. It should cover preparations, required network size, expected lifetime, and required practicalities.
Distributed and resource constrained nodes do not have existing debugging facilities.	Tailored testing tools should be implemented for the traceability of errors in compile- and run-time even in remote locations.
New deployment specific sensors and actuators require an iteration of tests.	As deployment preparations, a testing checklist should be used to ensure functioning of the previously tested components. Energy consumption should be checked after any modifications.
The data must be usable for the end-user.	Data must be interpret correctly and made available to the end-user, e.g. through several interfaces for integration. The network might be installed by the end-user, and a tailored installation tools may be required.
Deployment of thousands of nodes require tailored maintenance actions, since nodes eventually fill require replacing.	Maintenance tools are required. A bookkeeping with automated configuration of the nodes is required. An autonomous OTAP ensures that nodes get automatically newest software when deployed.
Deploying multiple field experiments require repetitive work.	A ready and thoroughly tested end-to-end architecture ensures minimal tailoring.

tailoring, works on all abstraction levels, and raises the abstraction of the application development with the PDL processes. In conclusion, the lessons of the field experiments show the importance of realizing the design characteristics of WSNs at all abstraction levels.

7. SUMMARY OF PUBLICATIONS

This chapter summaries the publications included in this thesis. None of the publications have been used previously as part of any other doctoral thesis. The contribution of the author of this thesis is described for each publication. The publications were written and published between 2009 and 2013.

Publication [P1] presents an operating system kernel, HybridKernel, for resource constrained WSN nodes. HybridKernel uses a pre-emptive kernel with an event-driven protothread extension to achieve small memory overhead and high real-time guarantees.

The author designed and implemented the HybridKernel and was the main author of the publication.

Publication [P2] presents a lightweight OTAP method, PIDP, for resource constrained WSN nodes. PIDP does not need a temporary storage memory and has a fall-back method for unsuccessful transmission. The protocol implementation was tested with 25 and 178 node networks and the power consumption was analyzed.

The author extended the original conference publication [155], added the design of the application dissemination and the operating system support, further analyzed the power consumption of the protocol, and presented the comparison to other proposals.

Publication [P3] presents several practical pilot studies in WSN application development. The lessons of the studies are presented and several methods are described for conducting WSN application pilots successfully.

The author was part of the research group that executed the pilot studies and designed and implemented the presented solutions and tools. The author was the main author of the publication.

Publication [P4] presents the abstraction levels of the WSN application development and surveys infrastructure abstractions. The publication gathers several open questions and presents design directions that could solve some of the proposed problems.

The survey was conducted by the author.

Publication [P5] presents a unified service access for WSNs called WSN OpenAPI. It is a set of XML specifications and interfaces that homogenize real-time and archived access to measurements and actuators, and allow alert generation. WSN OpenAPI is evaluated with implementation, tests, and use cases of heterogeneous technologies.

The author was part of the research group that designed the WSN Open API. The author gave ideas to the publication, revised the publication, and wrote the greenhouse monitoring use case.

Publication [P6] defines an embedded cloud for IoTs and presents an embedded cloud design. The embedded cloud shares and expand the resources of IoT devices, distributes processing, and provides an unified access for end-user applications. The design is confirmed with a prototype implementation that works even with resource constrained WSN nodes.

The embedded cloud design and the implementation were conducted by the author, and the author was the main author of the paper.

8. CONCLUSIONS

Constructing ubiquitous applications requires a combination of heterogeneous sensor technologies. Resource constrained WSNs are a key technology for such applications, which have the requirements of low cost, small size, distributed networking, and an autonomous, long life-time operation. WSNs need application development methods that realize these requirements efficiently. The main research challenge is to create abstractions that allow easier, error-free, portable, and faster application development with resource constrained WSNs.

This thesis answered the main research question, "*What abstractions are needed for the application development for the resource constrained WSNs?*" with an abstraction model. The model consists of three levels: the node, the network, and the infrastructure abstractions. The node abstractions have methods to execute tasks, update software, interact with HW, and communicate on the resource constrained node. The network abstractions have methods for data acquisition, service discovery, and distributed processing over the nodes in one WSN. The infrastructure abstractions homogenize several heterogeneous sensing technologies behind one unified interface.

The abstraction model answered the main research question and the derived question, "*How to divide the abstractions hierarchically and what are the responsibilities of each level?*". The remainder of the derived research questions were answered and verified with proofs of concept as follows.

- To execute application tasks efficiently, an OS kernel is needed that combines pre-emptive scheduling with low overhead tasks for easy multitasking without excessive resource consumption. The presented HybridKernel combined a pre-emptive kernel and a cooperative event-driven kernel in to the same design. HybridKernel allowed scalable memory and execution overheads for pre-emptive processes that ensure the scheduling of high priority tasks. The cooperative threads allowed the several application tasks and typical programming style of a desktop OS.

- To disseminate new software and applications, an OTAP method is needed that allows all of the parts of the software to be updated without the risk of permanent failure, but also allows efficient updating of the applications. The combination of PIDP and PDL was presented. PIDP allowed software fixes to the whole firmware with a reliable fall-back method. PDL allowed small overhead and node specific application dissemination on the WNS nodes.

- To homogenize the data and actuator accessing, a unification interface is needed that covers a wide range of data sources and actuators. The WSN OpenAPI unified sensor and actuator access and its network-node-measurement hierarchy was used as an information model in use cases for arbitrary data sources.

- To unify the functionality of a node, a network and an infrastructure for distributed processing, distributed middleware should abstract the data sources, actuators, and processing capabilities of heterogeneous sensing technologies to unified services on each level. In the presented embedded cloud, DiMiWa abstracted these capabilities as services and PDL allowed distributed processing by using the services in its processing. As a result, the embedded cloud extended the resources of the WSN nodes.

As future work, the components in the embedded cloud should be further advanced to realize its full potential: 1) an intelligent PDL broker is needed to arbitrate PDL processes over the connected devices, 2) PDL should cover a larger range of values efficiently, 3) the infrastructure in the embedded cloud should be distributed to avoid single points of failure. In conclusion, more research focus is needed on distributing processing over heterogeneous WSNs, as ambient intelligence requires data sharing, distributed processing, and collaborative decision making to work autonomously and reliably.

The results of this thesis have facilitated WSN application development at all abstraction levels and the lessons of the field experiments provided insight into abstraction and application development with WSNs. The proofs of concept were implemented on a resource constrained WSN node to verify their feasibility. The results will remain topical and valid in the future. Although the constant development of ICs will increase the resources of the current sized nodes, the same advances will produce smaller nodes that will remain resource constrained.

BIBLIOGRAPHY

[1] K. Aberer, M. Hauswirth, and A. Salehi, “Infrastructure for data processing in large-scale interconnected sensor networks,” in *Mobile Data Management, 2007 International Conference on*, may 2007, pp. 198 –205.

[2] A. Ahmed, H. Shi, and Y. Shang, “A survey on network protocols for wireless sensor networks,” in *Information Technology: Research and Education, 2003. Proceedings. ITRE2003. International Conference on*, Aug 2003, pp. 301–305.

[3] K. Akkaya and M. Younis, “A survey on routing protocols for wireless sensor networks,” *Ad Hoc Networks*, vol. 3, no. 3, pp. 325 – 349, 2005. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1570870503000738

[4] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, “Wireless sensor networks: a survey,” *Computer Networks*, vol. 38, no. 4, pp. 393 – 422, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/B6VRG-44W46D4-1/2/f18cba34a1b0407e24e97fa7918cdfdc

[5] I. Akyildiz, T. Melodia, and K. Chowdury, “Wireless multimedia sensor networks: A survey,” *Wireless Communications, IEEE*, vol. 14, no. 6, pp. 32–39, 2007.

[6] S. Alam, M. Chowdhury, and J. Noll, “Senaas: An event-driven sensor virtualization approach for internet of things cloud,” in *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, nov. 2010, pp. 1 –6.

[7] A. Alamri, W. S. Ansari, M. M. Hassan, M. S. Hossain, A. Alelaiwi, and M. A. Hossain, “A Survey on Sensor-Cloud: Architecture, Applications, and Approaches,” *International Journal of Distributed Sensor Networks*, vol. 2013, no. Article ID 917923, p. 18, 2013.

[8] D. Alessandrelli, M. Petracca, and P. Pagano, "T-res: enabling reconfigurable in-network processing in iot-based wsns," in *International Conference on Distributed Computing in Sensor Systems, DCOSS 2013. International Workshop on Internet of ThingsŮIdeas and Perspectives (IoTIP-13)*, 2013, pp. 337–345.

[9] T. Alliance, "Tinyos 2.1 adding threads and memory protection to tinyos," in *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. New York, NY, USA: ACM, 2008, pp. 413–414.

[10] I. Amundson, M. Kushwaha, X. Koutsoukos, S. Neema, and J. Sztipanovits, "Efficient integration of web services in ambient-aware sensor network applications," in *Broadband Communications, Networks and Systems, 2006. BROADNETS 2006. 3rd International Conference on*, 2006, pp. 1–8.

[11] "Arduino product documentation," Available: http://arduino.cc, Arduino, visited: May 07, 2012.

[12] M. Aslam, S. Rea, and D. Pesch, "Service provisioning for the wsn cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, june 2012, pp. 962 –969.

[13] Atmel, "Atmega128l data sheet," [ONLINE] http://www.atmel.com/dyn/resources/prod_documents/doc2467.pdf, 2010.

[14] ——, "Atmega1281 product page," [ONLINE] http://www.atmel.com/devices/atmega1281.aspx, 2014.

[15] ——, "Atmega2560 product page," [ONLINE] http://www.atmel.com/devices/atmega2560.aspx, 2014.

[16] ——, "Atmega328p product page," [ONLINE] http://www.atmel.com/devices/atmega328p.aspx, 2014.

[17] ——, "Atmel at86rf230 datasheet," [ONLINE] http://www.atmel.com/Images/doc5131.pdf, 2014.

[18] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer Networks*, vol. 54, no. 15, pp. 2787 – 2805, 2010. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128610001568

[19] E. Avilés-López and J. A. García-Macías, "Tinysoa: a service-oriented architecture for wireless sensor networks," *Service Oriented Computing*

and Applications, vol. 3, no. 2, pp. 99–108, 2009. [Online]. Available: http://dx.doi.org/10.1007/s11761-009-0043-x

[20] R. Barr, J. C. Bicket, D. S. Dantas, B. Du, T. W. D. Kim, B. Zhou, and E. G. Sirer, "On the need for system-level support for ad hoc and sensor networks," *SIGOPS Oper. Syst. Rev.*, vol. 36, no. 2, pp. 1–5, Apr. 2002. [Online]. Available: http://doi.acm.org/10.1145/509526.509528

[21] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. New York, NY, USA: ACM, 2008, pp. 43–56.

[22] V. Berzins, M. Gray, and D. Naumann, "Abstraction-based software development," *Commun. ACM*, vol. 29, no. 5, pp. 402–415, May 1986. [Online]. Available: http://doi.acm.org/10.1145/5689.5691

[23] J. Beutel, M. Dyer, R. Lim, C. Plessl, M. Wohrle, M. Yucel, and L. Thiele, "Automated wireless sensor network testing," in *Networked Sensing Systems, 2007. INSS '07. Fourth International Conference on*, June 2007, pp. 303–303.

[24] S. Bhatti, J. Carlson, H. Dai, and et al., "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, Aug. 2005.

[25] M. Botts, G. Percivall, C. Reed, and J. Davidson, "Ogc® sensor web enablement: Overview and high level architecture," in *GeoSensor Networks*, ser. Lecture Notes in Computer Science, S. Nittel, A. Labrinidis, and A. Stefanidis, Eds. Springer Berlin / Heidelberg, 2008, vol. 4540, pp. 175–190, 10.1007/978-3-540-79996-2_10. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-79996-2_10

[26] M. Botts and A. Robin, "Ogc® sensor model language (sensorml) implementation specification, version 1.0.0," *OpenGIS® Implementation Specification. Open Geospatial Consortium. OGC® 07-000*, 2007.

[27] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye, "A semantics-based middleware for utilizing heterogeneous sensor networks," *Distributed Computing in Sensor Systems*, pp. 174–188, 2007.

[28] A. Boulis, C.-C. Han, R. Shea, and M. B. Srivastava, "Sensorware: Programming sensor networks beyond code update and querying," *Pervasive and Mobile Computing*, vol. 3, no. 4, pp. 386 –

412, 2007, middleware for Pervasive Computing. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1574119207000314

[29] A. Bröring, J. Echterhoff, S. Jirka, I. Simonis, T. Everding, C. Stasch, S. Liang, and R. Lemmens, "New generation sensor web enablement," *Sensors*, vol. 11, no. 3, pp. 2652–2699, 2011.

[30] 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*, ser. SenSys '09. New York, NY, USA: ACM, 2009, pp. 169–182. [Online]. Available: http://doi.acm.org/10.1145/1644038.1644056

[31] S. Brown and C. J. Sreenan, "Software updating in wireless sensor networks: A survey and lacunae," *Journal of Sensor and Actuator Networks*, vol. 2, no. 4, pp. 717–760, 2013.

[32] A. Bröring, C. Stasch, and J. Echterhoff, "Ogc® sensor observation service interface standard, version 2.0," *OpenGIS® Implementation Standard. Open Geospatial Consortium. OGC 12-006*, 2012.

[33] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS operating system: Towards Unix-like abstractions for wireless sensor networks," in *Proceedings of the 7th international conference on Information processing in sensor networks*. IEEE Computer Society Washington, DC, USA, 2008, pp. 233–244.

[34] T. Cao, Q.and Abdelzaher, J. Stankovic, K. Whitehouse, and L. Luo, "Declarative tracepoints: a programmable and application independent debugging system for wireless sensor networks," in *Proceedings of the 6th ACM conference on Embedded network sensor systems*, ser. SenSys '08. New York, NY, USA: ACM, 2008, pp. 85–98. [Online]. Available: http://doi.acm.org/10.1145/1460412.1460422

[35] V. Casola, A. Gaglione, and A. Mazzeo, "A Reference Architecture for Sensor Networks Integration and Management," *GeoSensor Networks*, pp. 158–168, 2009.

[36] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon, "Retos: Resilient, expandable, and threaded operating system for wireless sensor networks," in *Information Processing in Sensor Networks, 2007. IPSN 2007. 6th International Symposium on*, April 2007, pp. 148–157.

[37] I. Chatzigiannakis, G. Mylonas, and S. Nikoletseas, “50 ways to build your application: A survey of middleware and systems for wireless sensor networks,” in *Emerging Technologies and Factory Automation, 2007. ETFA. IEEE Conference on*, Sept 2007, pp. 466–473.

[38] C.-Y. Chong and S. Kumar, “Sensor networks: evolution, opportunities, and challenges,” *Proceedings of the IEEE*, vol. 91, no. 8, pp. 1247–1256, Aug 2003.

[39] X. Chu, Kobialka, B. Durnota, and R. Buyya, “Open sensor web architecture: Core services,” in *Intelligent Sensing and Information Processing, 2006. ICISIP 2006. Fourth International Conference on*, 2006, pp. 98–103.

[40] C. code repository, “File: contiki-2.x/core/sys/mt.h, revision 1.6,” [ONLINE] http ://contiki.cvs.sourceforge.net/, Apr. 2009.

[41] ——, “File: contiki-2.x/core/sys/process.h, revision 1.16,” [ONLINE] http ://-contiki.cvs.sourceforge.net/, Apr. 2009.

[42] T. code repository, “File: tinyos-2.x/tos/lib/tosthreads/types/thread.h, revision 1.1,” [ONLINE] http ://tinyos.cvs.sourceforge.net/, Apr. 2009.

[43] W. Colitti, K. Steenhaut, and N. De Caro, “Integrating wireless sensor networks with the web,” *Extending the Internet to Low powerand Lossy Networks (IP+ SN 2011)*, 2011.

[44] P. Corke, T. Wark, R. Jurdak, W. Hu, P. Valencia, and D. Moore, “Environmental wireless sensor networks,” *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1903 –1917, nov. 2010.

[45] P. Costa, L. Mottola, A. Murphy, and G. Picco, “Programming wireless sensor networks with the teenylime middleware,” in *Middleware 2007*, ser. Lecture Notes in Computer Science, R. Cerqueira and R. Campbell, Eds. Springer Berlin Heidelberg, 2007, vol. 4834, pp. 429–449. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-76778-7_22

[46] S. Cox, “Observations and measurements - xml implementation, version 2.0,” *OpenGIS® Implementation standard. Open Geospatial Consortium. OGC 10-025r1*, 2011.

[47] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G. Picco, “Tinylime: bridging mobile and sensor networks through middleware,” in *Per-*

vasive Computing and Communications, 2005. PerCom 2005. Third IEEE International Conference on, 2005, pp. 61 – 72.

[48] R. Dasgupta, “Anatomy of rtos and analyze the best-fit for small, medium and large footprint embedded devices in wireless sensor network,” *Sensor Technologies and Applications, 2008. SENSORCOMM ’08. Second International Conference on*, pp. 598–603, Aug. 2008.

[49] I. Demirkol, C. Ersoy, and F. Alagoz, “Mac protocols for wireless sensor networks: a survey,” *Communications Magazine, IEEE*, vol. 44, no. 4, pp. 115–121, April 2006.

[50] Digi International Inc., “Xbee wireless rf modules,” Available: http://www.digi.com/xbee/, 2014.

[51] W. Dong, C. Chen, X. Liu, and J. Bu., “Providing os support for wireless sensor networks: Challenges and approaches,” *IEEE Communications Surveys and Tutorials*, vol. 12, no. 4, 2010.

[52] W. Dong, C. Chen, X. Liu, J. Bu, and Y. Gao, “A lightweight and density-aware reprogramming protocol for wireless sensor networks,” *Mobile Computing, IEEE Transactions on*, vol. 10, no. 10, pp. 1403–1415, 2011.

[53] W. Dong, Y. Liu, C. Chen, J. Bu, C. Huang, and Z. Zhao, “R2: Incremental reprogramming using relocatable code in networked embedded systems,” *Computers, IEEE Transactions on*, vol. 62, no. 9, pp. 1837–1849, 2013.

[54] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen, “Elon: Enabling efficient and long-term reprogramming for wireless sensor networks,” *SIGMETRICS Perform. Eval. Rev.*, vol. 38, no. 1, pp. 49–60, June 2010. [Online]. Available: http://doi.acm.org/10.1145/1811099.1811046

[55] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen, “R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems,” in *INFOCOM, 2013 Proceedings IEEE*, 2013, pp. 315–319.

[56] C. Duffy, U. Roedig, J. Herbert, and C. Sreenan, “An experimental comparison of event driven and multi-threaded sensor node operating systems,” *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops ’07. Fifth Annual IEEE International Conference on*, pp. 267–271, March 2007.

[57] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt, “Run-time dynamic linking for reprogramming wireless sensor networks,” in *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Boulder, Colorado, USA, Nov. 2006. [Online]. Available: http://www.sics.se/ adam/dunkels06runtime.pdf

[58] A. Dunkels, B. Grönvall, and T. Voigt, “Contiki - a lightweight and flexible operating system for tiny networked sensors,” in *Proc. 29th Annual IEEE International Conference on Local Computer Networks*, Tampa, FL, USA, Nov. 2004, pp. 455–462.

[59] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali, “Protothreads: Simplifying event-driven programming of memory-constraine embedded systems,” in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, Boulder, Colorado, USA, Nov. 2006, pp. 29–42.

[60] M. Dyer, J. Beutel, T. Kalt, P. Oehen, L. Thiele, K. Martin, and P. Blum, “Deployment support network - A toolkit for the development of WSNs,” *Wireless Sensor Networks*, pp. 195–211, 2007.

[61] B. Ebeling, S. Hoyer, and J. Bührig, “What are your favorite methods? - an examination on the frequency of research methods for is conferences from 2006 to 2010,” p. 200, 2012.

[62] J. Echterhoff and T. Everding, “Ogc® sensor event service interface specification (proposed),” *Candidate OpenGIS® Discussion paper Open Geospatial Consortium. OGC 08-133*, 2008.

[63] M. Eisenhauer, C. Prause, M. Jahn, and M. Jentsch, “Middleware for wireless devices and sensors - energy efficiency at device level,” in *Sensor Mesh and Ad Hoc Communications and Networks (SECON), 2010 7th Annual IEEE Communications Society Conference on*, 2010, pp. 1–3.

[64] M. Eisenhauer, P. Rosengren, and P. Antolin, “A development platform for integrating wireless devices and sensors into ambient intelligence systems,” in *Sensor, Mesh and Ad Hoc Communications and Networks Workshops, 2009. SECON Workshops ’09. 6th Annual IEEE Communications Society Conference on*, 2009, pp. 1–3.

[65] Energizer, “Energizer l91 ultimate lithium aa battery product datasheet,” [ONLINE] http://data.energizer.com/PDFs/l91.pdf, June 2014.

[66] A. Eswaran, A. Rowe, and R. Rajkumar, “Nano-rk: An energy-aware resource-centric rtos for sensor networks,” in *RTSS ’05: Proceedings of the 26th IEEE International Real-Time Systems Symposium.* Washington, DC, USA: IEEE Computer Society, 2005, pp. 256–265.

[67] R. Fielding, “Architectural styles and the design of network-based software architectures,” Ph.D. dissertation, University of California, Irvine, 2000. [Online]. Available: http://www.ics.uci.edu/ fielding/pubs/dissertation/top.htm

[68] FMI, “Ilmatieteen laitoksen avoin data,” [ONLINE] https://ilmatieteenlaitos.fi/avoin-data, Sept. 2013.

[69] C.-L. Fok, G.-C. Roman, and C. Lu, “Mobile agent middleware for sensor networks: an application case study,” in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, 2005, pp. 382 – 387.

[70] G. Fox, S. Kamburugamuve, and R. Hartman, “Architecture and measured characteristics of a cloud based internet of things,” in *Collaboration Technologies and Systems (CTS), 2012 International Conference on*, may 2012, pp. 6 –12.

[71] FreeRTOS, “Freertos homepage - coroutines explained,” [ONLINE] http://www.freertos.org/, 2010.

[72] 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*, ser. PLDI ’03. New York, NY, USA: ACM, 2003, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/781131.781133

[73] D. Gelernter, “Generative communication in linda,” *ACM Trans. Program. Lang. Syst.*, vol. 7, pp. 80–112, January 1985. [Online]. Available: http://doi.acm.org/10.1145/2363.2433

[74] A. Gómez-Goiri and D. López-de Ipiña, “A triple space-based semantic distributed middleware for internet of things,” in *Current Trends in Web Engineering*, ser. Lecture Notes in Computer Science, F. Daniel and F. Facca, Eds. Springer Berlin Heidelberg, 2010, vol. 6385, pp. 447–458. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-16985-4_43

[75] L. Gomez and A. Laube, "Ontological middleware for dynamic wireless sensor data processing," in *Sensor Technologies and Applications, 2009. SENSORCOMM '09. Third International Conference on*, 2009, pp. 145–151.

[76] W. Grosky, A. Kansal, S. Nath, J. Liu, and F. Zhao, "Senseweb: An infrastructure for shared sensing," *Multimedia, IEEE*, vol. 14, no. 4, pp. 8 –13, oct.-dec. 2007.

[77] L. Gu and J. A. Stankovic, "t-kernel: providing reliable os support to wireless sensor networks," in *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2006, pp. 1–14.

[78] S. Hadim and N. Mohamed, "Middleware: middleware challenges and approaches for wireless sensor networks," *Distributed Systems Online, IEEE*, vol. 7, no. 3, p. 1, march 2006.

[79] A. Hagedorn, D. Starobinski, and A. Trachtenberg, "Rateless deluge: Over-the-air programming of wireless sensor networks using random linear codes," in *Information Processing in Sensor Networks, 2008. IPSN '08. International Conference on*, 2008, pp. 457–466.

[80] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*. New York, NY, USA: ACM, 2005, pp. 163–176.

[81] M. M. Hassan, B. Song, and E.-N. Huh, "A framework of sensor-cloud integration opportunities and challenges," in *Proceedings of the 3rd International Conference on Ubiquitous Information Management and Communication*, ser. ICUIMC '09. New York, NY, USA: ACM, 2009, pp. 618–626. [Online]. Available: http://doi.acm.org/10.1145/1516241.1516350

[82] S. Havens, "Ogc® transducer markup language (tml) implementation specification, version 1.0.0, retired," *OpenGIS® Implementation Specification. Open Geospatial Consortium. OGC 06-010r6*, 2007.

[83] K. Henricksen and R. Robinson, "A survey of middleware for sensor networks: state-of-the-art and future directions," in *Proceedings of the international workshop on Middleware for sensor networks*, ser. MidSens '06. New York, NY, USA: ACM, 2006, pp. 60–65. [Online]. Available: http://doi.acm.org/10.1145/1176866.1176877

[84] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, “System architecture directions for networked sensors,” *SIGPLAN Not.*, vol. 35, no. 11, pp. 93–104, 2000.

[85] J. Honkola, H. Laine, R. Brown, and O. Tyrkko, “Smart-M3 information sharing platform,” in *Computers and Communications (ISCC), 2010 IEEE Symposium on*. IEEE, 2010, pp. 1041–1046.

[86] J. Hu, C. Xue, Y. He, and E.-M. Sha, “Reprogramming with minimal transferred data on wireless sensor network,” in *Mobile Adhoc and Sensor Systems, 2009. MASS ’09. IEEE 6th International Conference on*, 2009, pp. 160–167.

[87] W. Hu, T. Dinh, P. Corke, and s. Jha, “Design and deployment of long-term outdoor sensornets: Experiences from a sugar farm,” *Pervasive Computing, IEEE*, vol. PP, no. 99, pp. 1 –1, 2010.

[88] J. W. Hui, “Deluge 2.0 - tinyos network programming,” [Online]. Available: http://www.cs.berkeley.edu/ jwhui/deluge/deluge-manual.pdf, July 2005, [Accessed: Nov. 13, 2009].

[89] J. W. Hui and D. Culler, “The dynamic behavior of a data dissemination protocol for network programming at scale,” in *SenSys ’04: Proceedings of the 2nd international conference on Embedded networked sensor systems*. New York, NY, USA: ACM, 2004, pp. 81–94.

[90] *IEEE Standard for Information Technology—Telecommunications and Information Exchange Between Systems—Local and Metropolitan Area Networks—Specific Requirements—Part 15.4: Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPAN)*, Sept. 2006, IEEE Std 802.15.4-2006.

[91] IETF, “Ietf constrained application protocol (coap) working group,” [ONLINE] https://datatracker.ietf.org/doc/draft-ietf-core-coap/, Oct. 2013.

[92] IETF, J. Hui, and P. Thubert, “Compression format for ipv6 datagrams over ieee 802.15.4-based networks - rfc6282,” [ONLINE] http://tools.ietf.org/html/rfc6282, Oct. 2014.

[93] T. Instruments, “Ti msp430f15x, msp430f16x, msp430f161x data sheet,” [ONLINE] http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf, 2010.

[94] Intel, “Intel mote 2 engineering platform data sheet,” Available: http://wsn.cse.wustl.edu/images/c/cb/Imote2-ds-rev2_2.pdf, 2014.

[95] M. Jahn, M. Jentsch, C. Prause, F. Pramudianto, A. Al-Akkad, and R. Reiners, “The energy aware smart home,” in *Future Information Technology (FutureTech), 2010 5th International Conference on*, May 2010, pp. 1–8.

[96] Y.-J. Jeon, S.-H. Park, and J.-S. Park, “Sensor node middleware to support web-based applications over wireless sensor networks,” in *Local Computer Networks, 2009. LCN 2009. IEEE 34th Conference on*, 2009, pp. 963–970.

[97] J. Jeong and D. Culler, “Incremental network programming for wireless sensors,” in *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, 2004, pp. 25–33.

[98] Y. Jeong, E. Song, G. Chae, M. Hong, and D. Park, “Large-Scale Middleware for Ubiquitous Sensor Networks,” *Intelligent Systems, IEEE*, vol. 25, no. 2, pp. 48–59, 2010.

[99] S. Jirka, A. Bröring, and D. Nüst, “Ogc® sensor observable registry (sor) discussion paper,” *Candidate OpenGIS® Public Discussion paper Open Geospatial Consortium. OGC 09-112r1*, 2010.

[100] S. Jirka and D. Nüst, “Ogc® sensor instance registry discussion paper,” *Candidate OpenGIS® Discussion paper Open Geospatial Consortium. OGC 10-171*, 2010.

[101] J. Juntunen, M. Kuorilehto, M. Kohvakka, V. Kaseva, M. Hannikainen, and T. Hamalainen, “WSN API: Application programming interface for wireless sensor networks,” Sept. 2006, pp. 1–5.

[102] H. Karvonen, J. Suhonen, J. Petäjäjärvi, M. Hämäläinen, M. Hännikäinen, and A. Pouttu, “Hierarchical architecture for multi-technology wireless sensor networks for critical infrastructure protection,” *Wireless Personal Communications*, pp. 1–21, 2014. [Online]. Available: http://dx.doi.org/10.1007/s11277-014-1686-2

[103] V. Kaseva, T. D. Hämäläinen, and M. Hännikäinen, “A wireless sensor network for hospital security: from user requirements to pilot deployment,” *EURASIP J. Wirel. Commun. Netw.*, vol. 2011, pp. 17:1–17:15, January 2011. [Online]. Available: http://dx.doi.org/10.1155/2011/920141

[104] V. Kaseva, “Localization and time synchronization services for resource constrained wireless sensor networks,” *Tampereen teknillinen yliopisto. Julkaisu - Tampere University of Technology. Publication; 1051*, 2012.

[105] A. Khattak, L. T. Vinh, D. V. Hung, P. T. H. Truc, L. X. Hung, D. Guan, Z. Pervez, M. Han, S. Lee, and Y.-K. Lee, "Context-aware human activity recognition and decision making," in *e-Health Networking Applications and Services (Healthcom), 2010 12th IEEE International Conference on*, july 2010, pp. 112 –118.

[106] G. Kiczales, "Towards a new model of abstraction in software engineering," in *Object Orientation in Operating Systems, 1991. Proceedings., 1991 International Workshop on*, Oct 1991, pp. 127–128.

[107] T.-H. Kim and S. Hong, "State machine based operating system architecture for wireless sensor networks," vol. 3320, pp. 803–806, 2005. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-30501-9_158

[108] M. Kohvakka, "Medium access control and hardware prototype designs for low-energy wireless sensor networks," *Tampereen teknillinen yliopisto. Julkaisu - Tampere University of Technology. Publication; 808*, 2009.

[109] J. Koshy and R. Pandey, "Vmstar: Synthesizing scalable runtime environments for sensor networks," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '05. New York, NY, USA: ACM, 2005, pp. 243–254. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098945

[110] M. Kovatsch, M. Lanter, and S. Duquennoy, "Actinium: A restful runtime container for scriptable internet of things applications," in *Internet of Things (IOT), 2012 3rd International Conference on the*, Oct 2012, pp. 135–142.

[111] M. Kovatsch, S. Mayer, and B. Ostermaier, "Moving application logic from the firmware to the cloud: Towards the thin server architecture for the internet of things," in *Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on*, July 2012, pp. 751–756.

[112] L. Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Distributed Computing Systems Workshops, 2002. Proceedings. 22nd International Conference on*, 2002, pp. 575 – 578.

[113] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis, "Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea," in *Proceedings of the 3rd international*

conference on Embedded networked sensor systems, ser. SenSys '05. New York, NY, USA: ACM, 2005, pp. 64–75. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098926

[114] C. W. Krueger, "Software reuse," *ACM Comput. Surv.*, vol. 24, no. 2, pp. 131–183, June 1992. [Online]. Available: http://doi.acm.org/10.1145/130844.130856

[115] M. Kuorilehto, T. Alho, M. Hännikäinen, and T. D. Hämäläinen, "Sensoros: A new operating system for time critical wsn applications," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Heidelberg, Germany, Aug. 2007, pp. 431–442.

[116] M. Kuorilehto, M. Hännikäinen, and T. D. Hämäläinen, "A survey of application distribution in wireless sensor networks," *EURASIP J. Wirel. Commun. Netw.*, vol. 2005, no. 5, pp. 774–788, Oct. 2005. [Online]. Available: http://dx.doi.org/10.1155/WCN.2005.774

[117] M. Kuorilehto, M. Kohvakka, J. Suhonen, P. Hämäläinen, M. Hännikäinen, and T. D. Hämäläinen, *Ultra-Low Energy Wireless Sensor Networks in Practice - Theory, Realization and Deployment*. John Wiley & Sons, Ltd., 2007.

[118] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, apr. 2006, p. 8 pp.

[119] S.-Y. Lau, T.-H. Chang, S.-Y. Hu, H.-J. Huang, L. de Shyu, C.-M. Chiu, and P. Huang, "Sensor networks for everyday use: the bl-live experience," in *Sensor Networks, Ubiquitous, and Trustworthy Computing, 2006. IEEE International Conference on*, vol. 1, 2006, p. 7 pp.

[120] X. H. Le, S. Lee, P. T. H. Truc, L. T. Vinh, A. Khattak, M. Han, D. V. Hung, M. Hassan, M. Kim, K.-H. Koo, Y.-K. Lee, and E.-N. Huh, "Secured wsn-integrated cloud computing for u-life care," in *Consumer Communications and Networking Conference (CCNC), 2010 7th IEEE*, 2010, pp. 1–2.

[121] K. Lee, D. Murray, D. Hughes, and W. Joosen, "Extending sensor networks into the cloud using amazon web services," in *Networked Embedded Systems for Enterprise Applications (NESEA), 2010 IEEE International Conference on*, 2010, pp. 1–7.

[122] T. Leppänen, J. A. Lacasia, A. Ramalingam, M. Liu, E. Harjula, P. Närhi, J. Ylioja, J. Riekki, K. Sezaki, Y. Tobe, and T. Ojala, "Interoperable mobile agents in heterogeneous wireless sensor networks," in *Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems*, ser. SenSys '13. New York, NY, USA: ACM, 2013, pp. 64:1–64:2. [Online]. Available: http://doi.acm.org/10.1145/2517351.2517382

[123] T. Leppänen, M. Liu, E. Harjula, A. Ramalingam, J. Ylioja, P. Narhi, J. Riekki, and T. Ojala, "Mobile agents for integration of internet of things and wireless sensor networks," in *Proceedings of the 2013 IEEE International Conference on Systems, Man, and Cybernetics*, ser. SMC '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 14–21. [Online]. Available: http://dx.doi.org/10.1109/SMC.2013.10

[124] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," in *ACM Sigplan Notices*, vol. 37, no. 10. ACM, 2002, pp. 85–95.

[125] P. Levis, "Tinyos programming," [ONLINE] http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf, Mar. 2014.

[126] P. Levis, N. Lee, M. Welsh, and D. Culler, "Tossim: Accurate and scalable simulation of entire tinyos applications," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 126–137. [Online]. Available: http://doi.acm.org/10.1145/958491.958506

[127] S. Li, Y. Lin, S. Son, J. Stankovic, and Y. Wei, "Event detection services using data service middleware in distributed sensor networks," *Telecommunication Systems*, vol. 26, no. 2, pp. 351–368, 2004.

[128] S. Li, S. Son, and J. Stankovic, "Event detection services using data service middleware in distributed sensor networks," in *Information Processing in Sensor Networks*, ser. Lecture Notes in Computer Science, F. Zhao and L. Guibas, Eds. Springer Berlin Heidelberg, 2003, vol. 2634, pp. 502–517. [Online]. Available: http://dx.doi.org/10.1007/3-540-36978-3_34

[129] *Waspmote - Datasheet*, Available: http://www.libelium.com/waspmote, Libelium Comunicaciones Distribuidas S.L., 2011, document version: v1.4 - 10/2011.

[130] Y. Lim and J. Park, “Sensor Resource Sharing Approaches in Sensor-Cloud Infrastructure,” *International Journal of Distributed Sensor Networks*, vol. 2014, no. Article ID 476090, p. 8, 2014.

[131] T. Liu and M. Martonosi, “Impala: a middleware system for managing autonomic, parallel sensor systems,” *SIGPLAN Not.*, vol. 38, pp. 107–118, June 2003. [Online]. Available: http://doi.acm.org/10.1145/966049.781516

[132] X. Liu, K. M. Hou, C. D. Vaulx, C. Guo, H. Shi, and B. Tian, “Hybrid real-time operating system for resource-constraint wireless sensor nodes,” *Journal of Software*, vol. 9, no. 7, 2014. [Online]. Available: http://ojs.academypublisher.com/index.php/jsw/article/view/jsw090717671780

[133] K. Lorincz, B.-r. Chen, J. Waterman, G. Werner-Allen, and M. Welsh, “Resource aware programming in the pixie os,” in *SenSys ’08: Proceedings of the 6th ACM conference on Embedded network sensor systems*. New York, NY, USA: ACM, 2008, pp. 211–224.

[134] Q. Luo, H. Wu, W. Xue, and B. He, “Benchmarking in-network sensor query processing,” *Technical report - The Hong Kong University of Science and Technology*, 2005.

[135] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, “Tag: a tiny aggregation service for ad-hoc sensor networks,” *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[136] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, “Tinydb: an acquisitional query processing system for sensor networks,” *ACM Trans. Database Syst.*, vol. 30, pp. 122–173, March 2005. [Online]. Available: http://doi.acm.org/10.1145/1061318.1061322

[137] S. Madria, V. Kumar, and R. Dalvi, “Sensor cloud: A cloud of virtual sensors,” *Software, IEEE*, vol. PP, no. 99, pp. 1–1, 2013.

[138] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, “Wireless sensor networks for habitat monitoring,” in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, ser. WSNA ’02. New York, NY, USA: ACM, 2002, pp. 88–97. [Online]. Available: http://doi.acm.org/10.1145/570738.570751

[139] D. Manjunath, “A review of current operating systems for wireless sensor networks,” in *22nd International Conference on Computers and Their Applica-*

tions, CATA-2007, Honolulu, Hawaii, USA, March 28-30, 2007, 2007, pp. 387–394.

[140] P. Marrón, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel, "Flexcup: A flexible and efficient code update mechanism for sensor networks," in *Wireless Sensor Networks*, ser. Lecture Notes in Computer Science, K. Römer, H. Karl, and F. Mattern, Eds. Springer Berlin Heidelberg, 2006, vol. 3868, pp. 212–227. [Online]. Available: http://dx.doi.org/10.1007/11669463_17

[141] W. Masri and Z. Mammeri, "Middleware for wireless sensor networks: A comparative analysis," in *Network and Parallel Computing Workshops, 2007. NPC Workshops. IFIP International Conference on*, sept. 2007, pp. 349 –356.

[142] G. McFerren, D. Hohls, G. Fleming, and T. Sutton, "Evaluating sensor observation service implementations," in *Geoscience and Remote Sensing Symposium,2009 IEEE International,IGARSS 2009*, vol. 5, 2009, pp. V–363–V–366.

[143] *IRIS Wireless Measurement System*, MEMSIC Inc., document: 6020-0124-02 Rev A.

[144] *MICAz Wireless Measurement System*, MEMSIC Inc., document: 6020-0065-05 Rev A.

[145] *TELOSB Mote Platform*, MEMSIC Inc., document: 6020-0094-04 Rev B.

[146] Memsic Inc., "Mcs410 cricket wireless location system - product specification," Available: http://www.memsic.com/userfiles/files/Datasheets/WSN/6020-0063-03_a_mcs410_cricket-t.pdf, Nov. 2014.

[147] Microchip, "Mplab c compiler for pic18 mcus," [ONLINE] http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&-nodeId=1406&dDocName=en010014, 2010.

[148] ——, "Pic18f8722 family data sheet," [ONLINE] http://ww1.microchip.com/downloads/en/DeviceDoc/39646c.pdf, 2010.

[149] R. Müller, M. Fabritius, and M. Mock, "An ogc compliant sensor observation service for mobile sensors," in *Advancing Geoinformation Science for a Changing World*, ser. Lecture Notes in Geoinformation and Cartography, S. Geertman, W. Reinhardt, and F. Toppen, Eds. Springer Berlin Heidelberg,

2011, pp. 163–184. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19789-5_9

[150] N. Mohamed and J. Al-Jaroodi, "Service-oriented middleware approaches for wireless sensor networks," in *System Sciences (HICSS), 2011 44th Hawaii International Conference on*, jan. 2011, pp. 1 –9.

[151] M. Molla and S. Ahamed, "A survey of middleware for sensor network and challenges," in *Parallel Processing Workshops, 2006. ICPP 2006 Workshops. 2006 International Conference on*, 2006, pp. 6 pp. –228.

[152] J. D. Mooney, "Bringing portability to the software process," *Dept. of Statistics and Comp. Sci., West Virginia Univ., Morgantown WV*, 1997.

[153] J. Mooney, "Strategies for supporting application portability," *Computer*, vol. 23, no. 11, pp. 59–70, Nov 1990.

[154] L. Mottola and G. P. Picco, "Programming wireless sensor networks: Fundamental concepts and state of the art," *ACM Comput. Surv.*, vol. 43, no. 3, pp. 19:1–19:51, Apr. 2011. [Online]. Available: http://doi.acm.org/10.1145/1922649.1922656

[155] L. Määttä, J. Suhonen, T. Laukkarinen, M. Hännikäinen, and T. D. Hämäläinen, "Program image dissemination protocol for low-energy multihop wireless sensor networks," in *International Symposium on System-on-Chip 2010*, Tampere, Finland, Sept. 2010, pp. 129–135.

[156] A. Murphy and W. Heinzelman, "Milan: Middleware linking applications and networks," *University of Rochester, Tech. Rep. TR-795*, 2002.

[157] A. Murphy, G. Picco, and G. Roman, "Lime: a middleware for physical and logical mobility," in *Distributed Computing Systems, 2001. 21st International Conference on.*, Apr 2001, pp. 524–533.

[158] S. Nath, J. Liu, and F. Zhao, "Challenges in building a portal for sensors worldwide," in *In First Workshop on WorldSensor-Web, Boulder,CO*. ACM, 2006, pp. 3–4.

[159] Nordic Semiconductor, "Single chip 2.4 GHz transceiver nrf24l01 - product specification," Available: http://www.nordicsemi.com/eng/Products/2.4GHz-RF/nRF24L01, Sept. 2006, revision: 1.0.

[160] ——, “Single chip 433/868/915 MHz transceiver nRF905 - product specification,” Available: http://www.nordicsemi.com/eng/Products/Sub-1-GHz-RF/nRF905, June 2006, revision: 1.4.

[161] J. Onkila, “Käyttöjärjestelmän edut langattomassa anturiverkossa,” *Tampere University of Technology - Theses*, 2011.

[162] T. O’Reilly, “Ogc® puck protocol standard version 1.4,” *Candidate OGC® Standard. OGC 09-127r2*, 2010.

[163] S. N. Pakzad, G. L. Fenves, S. Kim, and D. E. Culler, “Design and implementation of scalable wireless sensor network for structural monitoring,” *Journal of Infrastructure Systems*, vol. 14, no. 1, pp. 89–101, 2008. [Online]. Available: http://link.aip.org/link/?QIS/14/89/1

[164] R. K. Panta, S. Bagchi, and S. P. Midkiff, “Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation,” in *Proceedings of the 2009 Conference on USENIX Annual Technical Conference*, ser. USENIX’09. Berkeley, CA, USA: USENIX Association, 2009, pp. 32–32. [Online]. Available: http://dl.acm.org/citation.cfm?id=1855807.1855839

[165] R. Panta, I. Khalil, and S. Bagchi, “Stream: Low overhead wireless reprogramming for sensor networks,” in *INFOCOM 2007. 26th IEEE International Conference on Computer Communications. IEEE*, 2007, pp. 928–936.

[166] P. Parwekar, “From internet of things towards cloud of things,” in *Computer and Communication Technology (ICCCT), 2011 2nd International Conference on*, sept. 2011, pp. 329 –333.

[167] B. Pasztor and P. Hui, “Osone: A distributed operating system for energy efficient sensor network,” in *Teletraffic Congress (ITC), 2013 25th International*, 2013, pp. 1–9.

[168] G. P. Picco, “Software engineering and wireless sensor networks: Happy marriage or consensual divorce?” in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research*, ser. FoSER ’10. New York, NY, USA: ACM, 2010, pp. 283–286. [Online]. Available: http://doi.acm.org/10.1145/1882362.1882421

[169] M. E. Poorazizi, S. H. L. Liang, and A. J. S. Hunter, “Testing of sensor observation services: A performance evaluation,” in *Proceedings of the*

First ACM SIGSPATIAL Workshop on Sensor Web Enablement, ser. SWE '12. New York, NY, USA: ACM, 2012, pp. 32–38. [Online]. Available: http://doi.acm.org/10.1145/2451716.2451721

[170] G. Prakash, M. Thejaswini, M. SH, V. KR, and P. LM, "Energy efficient in-network data processing in sensor networks," *World Academy of Science, Engineering and Technology 48 2008*, vol. 48, 2008.

[171] Raspberry Pi, "Raspberry pi an arm gnu/linux box for 25$," Available: http://www.raspberrypi.org/, 2014.

[172] *Shimmer User Manual*, Realtime technologies Ltd., 2011, revision 2R.d.

[173] A. Rezgui and M. Eltoweissy, "Service-oriented sensor-actuator networks [ad hoc and sensor networks]," *Communications Magazine, IEEE*, vol. 45, no. 12, pp. 92–100, December 2007.

[174] ——, "Service-oriented sensor-actuator networks: Promises, challenges, and the road ahead," *Computer Communications*, vol. 30, no. 13, pp. 2627 – 2648, 2007, sensor-Actuated Networks {SANETs}. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0140366407002472

[175] K. Römer and M. Ringwald, "Increasing the visibility of sensor networks with passive distributed assertions," in *Proceedings of the workshop on Real-world wireless sensor networks*, ser. REALWSN '08. New York, NY, USA: ACM, 2008, pp. 36–40. [Online]. Available: http://doi.acm.org/10.1145/1435473.1435484

[176] K. Römer, "Programming paradigms and middleware for sensor networks," in *GI/ITG Workshop on Sensor Networks*, 2004, pp. 49–54.

[177] M. Rossi, N. Bui, G. Zanca, L. Stabellini, R. Crepaldi, and M. Zorzi, "Synapse++: Code dissemination in wireless sensor networks using fountain codes," *Mobile Computing, IEEE Transactions on*, vol. 9, no. 12, pp. 1749–1765, 2010.

[178] M. Rossi, G. Zanca, L. Stabellini, R. Crepaldi, A. Harris, and M. Zorzi, "Synapse: A network reprogramming protocol for wireless sensor networks using fountain codes," in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON '08. 5th Annual IEEE Communications Society Conference on*, 2008, pp. 188–196.

[179] B. Rubio, M. Diaz, and J. Troya, "Programming approaches and challenges for wireless sensor networks," in *Systems and Networks Communications, 2007. ICSNC 2007. Second International Conference on*, Aug 2007, pp. 36–36.

[180] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap) - rfc 7252," [ONLINE] http://www.rfc-editor.org/info/rfc7252, June 2014.

[181] C.-C. Shen, C. Srisathapornphat, and C. Jaikaeo, "Sensor information networking architecture and applications," *Personal Communications, IEEE*, vol. 8, no. 4, pp. 52 –59, Aug. 2001.

[182] J. Shneidman, P. Pietzuch, J. Ledlie, M. Roussopoulos, M. Seltzer, and M. Welsh, "Hourglass: An infrastructure for connecting sensor networks and applications," Harvard University, Tech. Rep., 2004, harvard Technical Report TR-21-04.

[183] I. Simonis, "Ogc® sensor alert service candidate implementation specification, version 0.9," *Candidate OpenGIS® Implementation Specification Open Geospatial Consortium. OGC 06-028r3*, 2006.

[184] ——, "Ogc® sensor web enablement architecture, version 0.4.0," *OGC Best Practice. Open Geospatial Consortium. OGC® 06-021r4*, 2008.

[185] I. Simonis and J. Echterhoff, "Ogc® sensor planning service implementation standard, version 2.0," *OpenGIS® Implementation Standard. Open Geospatial Consortium. OGC 09-000*, 2011.

[186] I. Simonis and A. Wytzisk, "Web notification service, version 0.1.0," *OpenGIS Discussion Paper. Open Geospatial Consortium. OGC 03-008r2*, 2003.

[187] 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*, vol. 10, no. 1, pp. 37–44, 2006.

[188] P. Spiess, S. Karnouskos, D. Guinard, D. Savio, O. Baecker, L. Souza, and V. Trifa, "Soa-based integration of the internet of things in enterprise services," in *Web Services, 2009. ICWS 2009. IEEE International Conference on*, 2009, pp. 968–975.

[189] STMicroelectronics, "Stm32 l0 series of ultra-low-power mcus," Available: http://www.st.com/web/en/catalog/mmc/SC1169/SS1817, 2014.

[190] G. Strazdins, A. Elsts, K. Nesenbergs, and L. Selavo, “Wireless sensor network operating system design rules based on real-world deployment survey,” *Journal of Sensor and Actuator Networks*, vol. 2, no. 3, pp. 509–556, 2013.

[191] R. Sugihara and R. K. Gupta, “Programming models for sensor networks: A survey,” *ACM Trans. Sen. Netw.*, vol. 4, pp. 8:1–8:29, April 2008. [Online]. Available: http://doi.acm.org/10.1145/1340771.1340774

[192] J. Suhonen, “Designs for the quality of service support in low-energy wireless sensor network protocols,” *Tampereen teknillinen yliopisto. Julkaisu - Tampere University of Technology. Publication; 1061*, 2012.

[193] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler, “Lessons from a sensor network expedition,” *Wireless Sensor Networks*, pp. 307–322, 2004.

[194] A. Tamayo, C. Granell, and J. Huerta, “Using swe standards for ubiquitous environmental sensing: A performance analysis,” *Sensors*, vol. 12, no. 9, pp. 12 026–12 051, 2012. [Online]. Available: http://www.mdpi.com/1424-8220/12/9/12026

[195] A. Tamayo, P. Viciano, C. Granell, and J. Huerta, “Empirical study of sensor observation services server instances,” in *Advancing Geoinformation Science for a Changing World*, ser. Lecture Notes in Geoinformation and Cartography, S. Geertman, W. Reinhardt, and F. Toppen, Eds. Springer Berlin Heidelberg, 2011, pp. 185–209. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-19789-5_10

[196] Tampere University of Technology, “Wsn openapi technical specification,” *WSN OpenAPI Technical Specification - r1.00-2013 - 2013-10-14 - Editor: Jukka Suhonen*, 2013. [Online]. Available: http://www.tkt.cs.tut.fi/research/gwg/downloads/WSN_OpenAPI_Specification_r1.0.pdf

[197] L. Tan and N. Wang, “Future internet: The internet of things,” in *Advanced Computer Theory and Engineering (ICACTE), 2010 3rd International Conference on*, vol. 5, aug. 2010, pp. V5–376 –V5–380.

[198] T. Tavares, R. Santana, M. Santana, and J. Estrella, “Performance evaluation on opengis consortium for sensor web enablement services,” in *ICSNC 2013, The Eighth International Conference on Systems and Networks Communications*, 2013, pp. 135–140.

[199] Texas Instruments Inc., “Cc1101 low-power sub-1ghz rf transceiver,” Available: http://www.ti.com/product/cc1101, 2014.

[200] ——, “CC2420 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF transceiver,” Available: http://www.ti.com/product/cc2420, 2014.

[201] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, “A macroscope in the redwoods,” in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, ser. SenSys ’05. New York, NY, USA: ACM, 2005, pp. 51–63. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098925

[202] N. Tsiftes, A. Dunkels, and T. Voigt, “Efficient sensor network reprogramming through compression of executable modules,” in *Sensor, Mesh and Ad Hoc Communications and Networks, 2008. SECON ’08. 5th Annual IEEE Communications Society Conference on*, 2008, pp. 359–367.

[203] N. Tsiftes, A. Dunkels, H. Zhitao, and T. Voigt, “Enabling large-scale storage in sensor networks with the coffee file system,” in *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, ser. IPSN ’09. Washington, DC, USA: IEEE Computer Society, 2009, pp. 349–360. [Online]. Available: http://portal.acm.org/citation.cfm?id=1602165.1602197

[204] D. Villa, F. Moya, F. J. Villanueva, O. Aceña, and J. C. López, “Ubiquitous virtual private network: A solution for wsn seamless integration,” *Sensors*, vol. 14, no. 1, pp. 779–794, 2014.

[205] Virtenio, “Innovative 2.4 ghz radio module for preon32-series,” Available: http://www.virtenio.com/en/products/radio-module.html, 2014.

[206] W3C, “Owl web ontology language reference,” [ONLINE] http://www.w3.org/TR/owl-ref/, Sept. 2012.

[207] ——, “Web services description language (wsdl) version 2.0 part 1: Core language,” [ONLINE] http://www.w3.org/TR/wsdl20/, Jan. 2013.

[208] W3C Incubator Group, “Semantic sensor network xg final report - w3c incubator group report 28 june 2011,” Available: http://www.w3.org/2005/Incubator/ssn/XGR-ssn-20110628/, 2014.

[209] W3C Working Group, "Web services architecture note 11 february 2004," Available: http://www.w3.org/TR/2004/NOTE-ws-arch-20040211/, 2014.

[210] ——, "Web services glossary note 11 february 2004," Available: http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/, 2014.

[211] M. Wang, J. Cao, J. Li, and S. Dasi, "Middleware for wireless sensor networks: A survey," *Journal of computer science and technology*, vol. 23, no. 3, pp. 305–326, 2008.

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

[213] M. K. Watfa and M. Moubarak, "A benchmarking tool for wireless sensor network embedded operating systems," *Journal of Networks*, vol. 9, no. 8, 2014. [Online]. Available: http://www.ojs.academypublisher.com/index.php/jnw/article/view/jnw090819711984

[214] M. Weiser, "The computer for the 21st century," *Scientific american*, vol. 265, no. 3, pp. 94–104, 1991.

[215] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 18 – 25, 2006.

[216] G. Werner-Allen, P. Swieskowski, and M. Welsh, "Motelab: a wireless sensor network testbed," in *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, April 2005, pp. 483–488.

[217] J. Yang, M. L. Soffa, L. Selavo, and K. Whitehouse, "Clairvoyant: a comprehensive source-level debugger for wireless sensor networks," in *Proceedings of the 5th international conference on Embedded networked sensor systems*, ser. SenSys '07. New York, NY, USA: ACM, 2007, pp. 189–203. [Online]. Available: http://doi.acm.org/10.1145/1322263.1322282

[218] J. Yannakopoulos and A. Bilas, "Cormos: a communication-oriented runtime system for sensor networks," in *Proc. 2nd European Workshop on Wireless Sensor Networks*, Istanbul, Turkey, Feb. 2005, pp. 342–353.

[219] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD record*, vol. 31, no. 3, pp. 9–18, 2002.

[220] J. Yi, S. Heu, B. Choi, H. Kim, H. Sue, and J. Kim, "TMO-NanoQ+: A Real-Time Kernel for Sensor Networks Supporting Time-Triggered and Message-Triggered Tasks," in *10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2007. ISORC'07*, 2007, pp. 228–235.

[221] J. Yick, B. Mukherjee, and D. Ghosal, "Wireless sensor network survey," *Computer Networks*, vol. 52, no. 12, pp. 2292 – 2330, 2008. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1389128608001254

[222] D. Yin and Y. Fang, "From sensor net to sensor grid a survey and taxonomy on sensor web," in *Geoscience and Remote Sensing Symposium, 2007. IGARSS 2007. IEEE International*, july 2007, pp. 2935 –2938.

[223] H. ying Zhou and K. mean Hou, "Limos: A lightweight multi-threading operating system dedicated to wireless sensor networks," *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pp. 3051–3054, Sept. 2007.

[224] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure - physical sensor management with virtualized sensors on cloud computing," in *Network-Based Information Systems (NBiS), 2010 13th International Conference on*, sept. 2010, pp. 1 –8.

[225] M. Yuriyama, T. Kushida, and M. Itakura, "A new model of accelerating service innovation with sensor-cloud infrastructure," in *SRII Global Conference (SRII), 2011 Annual*, 29 2011-april 2 2011, pp. 308 –314.

[226] H.-Y. Zhou, K.-M. Hou, J.-P. Chanet, C. de Vaulx, and G. De Sousa, "Limos: A tiny real-time micro-kernel for wireless objects," *Wireless Communications, Networking and Mobile Computing, 2006. WiCOM 2006.International Conference on*, pp. 1–4, Sept. 2006.

[227] B. P. . E. Zurich, "BTnode rev3 hardware reference," Available: http://www.btnode.ethz.ch/Documentation/BTnodeRev3HardwareReference, 2007, visited: May 04, 2012.

PUBLICATIONS

PUBLICATION 1

T. Laukkarinen, V. A. Kaseva, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "HybridKernel: Preemptive Kernel with Event-driven Extension for Resource Constrained Wireless Sensor Networks", *IEEE Workshop on Signal Processing Systems*, October 7–9, 2009, Tampere, Finland. doi:10.1109/SIPS.2009.5336243

HYBRIDKERNEL: PREEMPTIVE KERNEL WITH EVENT-DRIVEN EXTENSION FOR RESOURCE CONSTRAINED WIRELESS SENSOR NETWORKS

Teemu Laukkarinen, Ville A. Kaseva, Jukka Suhonen, Timo D. Hämäläinen, Marko Hännikäinen

{teemu.laukkarinen, ville.a.kaseva, jukka.suhonen, timo.d.hamalainen, marko.hannikainen}@tut.fi
Tampere University of Technology, Department of Computer Systems
P.O.Box 553, FI-33101, Tampere, Finland

ABSTRACT

A low-power wireless sensor network (WSN) implements dynamic communication protocols and embedded sensing applications on resource constrained platform. WSNs utilize dozens of tasks, which have differentiated realtime requirements. This requires an efficient implementation with the use of a real-time operating system optimized for WSNs. Current WSN operating systems are based either on preemptive or event-driven kernels. Preemption provides accurate timings but requires large data memory footprint. Event-driven kernels have small footprint but do not support time as accurately. This paper presents a new HybridKernel for WSNs which combines the advantages of both kernels. It meets five key requirements without any major drawbacks: it halves footprint of preemptive kernels, it provides 2 μs timing accuracy, it minimizes energy consumption, and it can be easily configured and used between preemptive and event-driven parts through a coherent system call interface.

Index Terms— wireless sensor networks, operating system kernels, embedded operating systems

1. INTRODUCTION

Wireless Sensor Networks (WSN) are expected to become key building blocks for ubiquitous smart environments [1]. A low-power WSN combines communication, sensing, and data processing with resource constrained platform, enabling its integration into numerous things [2]. Applications can be anything from various fields, such as automation, signal processing or raw data gathering. Real Time Operating Systems (RTOS) are utilized to serve the complex requirements of the WSN protocol implementations [3]. In general, the WSN RTOSes provide multitasking, deadline guarantee, resource sharing, and realtime reactiveness with coherent system call interface.

Current WSN RTOS implementatios use two different kernel types, either a preemptive multithreading kernel or an event-driven kernel [4]. The preemptive kernel provides accurate timings and a familiar system call interface for the application programmer. The preemptive threads have a context and they can be switched forcibly by the scheduler. Each context requires quite much data memory and context switching consumes processing time. These cause unwanted energy consumption and data memory overhead.

The event-driven kernels require less data memory and do not have the context switching overhead. Thus, they are widely used for WSNs. The kernel executes event handlers, which are associated to periodic and sporadic events. The event handlers are executed cooperatively, which enables fast context-free switching. However, the event handlers have to be executed to the completion. Thus, they cannot perform lengthy computations to guarantee fairness for others. This affects to the timing accuracy and reactiveness [3] [4]. Also, the application programmer has to be aware of this, which makes application programming slower and more difficult than with preemptive kernels [5].

In this paper, we propose HybridKernel, which implements a flexible event-driven extension on top of a preemptive kernel. A state-machine abstraction called protothreads [5] is used for the event-driven extension. The protothreads remove the slow and complex programming approach of the event handlers. HybridKernel is the first WSN RTOS kernel that has a coherent API between the preemptive and event-driven parts. It is scalable, it provides accurate timings, it improves energy efficiency, and it has small data memory footprint as shown in subsequent sections. The novel HybridKernel approach resolves problems with traditional preemptive and event-driven kernels.

The related work is presented in section 2. Section 3. describes the design of HybridKernel. Section 4. gives the implementation on Microchip PIC18F8722 microcontroller unit (MCU). Section 5. evaluates design and implementation. Conclusions and the future work are presented in section 6.

2. RELATED WORK

Event-driven kernels are a common solution for the small footprint WSN hardware platforms, which have 8-bit MCU with 4-64 KB of data memory. The preemptive kernels are considered feasible for the large footprint hardware platforms, such as 16 or 32bit MCUs with over 640 KB of data memory [4].

An experimental comparison between the event-driven kernel and preemptive kernel on WSNs is given in [6]. TinyOS [7] is used for the event-driven kernel and MantisOS [8] is used for the preemptive kernel. The results show that TinyOS manages to be more energy efficient at the high static traffic loads, while MantisOS suffers from context switching overhead. However, MantisOS manages better sporadic traffic loads, due to its preemptive scheduling. As a result, there is no completely adequate RTOS for the WSNs in small footprint platforms.

Three WSN RTOSes have introduced compromises between the two kernel types. Preemptive threads called TOSThreads were introduced for TinyOS [9]. In this approach, event-driven TinyOS is executed in one high priority preemptive thread and the applications are executed in lower priority threads. This approach guarantees execution time for TinyOS. The context overhead remains since the application threads can only use the event-driven kernel through a blocking system call API. Thus, one application requires one thread and context. The application programmer can control overhead by creating applications for the TinyOS kernel also. However, this adds unnecessary complexity to the application development.

Contiki implements preemptive multithreading support as an application library on top of the event-driven kernel [10]. As with TinyOS the context overhead remains. Contiki supports event-handler application execution in parallel with preemptive threads. Thus, the overhead can be controlled by the application programmer. However, the application development complexity increases.

LIMOS approach is to associate a set of preemptive threads to events [11]. Basically it is an event-driven kernel, which executes multiple preemptive threads in every event handler. As usual the event handlers have to run to the completion. This approach does not preserve strict timings between the event handlers. If every event handler has two preemptive threads, all the event handlers require two stacks. This increases the data memory overhead to the level of traditional preemptive kernel. However, the application programmer can avoid unnecessary context switches.

The protothreads are introduced in [5]. Long processing in event-handlers is usually split in phases with state-machines. Dunkels et al. have shown that using the protothreads as an abstraction for state-machines, code complexity decreases. The protothread API can be formed to resemble traditional preemptive multithreading API, which is familiar to the application programmers. The overhead of the protothreads is small increase in the execution time and the code size. Also, local variables are not saved across the blocking protothread system calls, which application programmer has to be aware of.

When compared to the traditional preemptive and event-driven kernels, HybridKernel proposes a kernel design that is scalable, provides timing accuracy and reactiveness, and is suitable for the small WSN platforms. HybridKernel supports both methods natively and through the coherent API, which Contiki and TinyOS do not provide. HybridKernel utilizes the protothreads for the event-driven extension to overcome difficult event handler programming. Use of the protothreads is more free with HybridKernel than traditional event-driven kernels, since the protothreads block each other only inside that very preemptive thread. Unlike LIMOS, HybridKernel minimizes both context overheads, since it minimizes the need for the preemptive threads. This reduces amount of context switching and needed data memory.

3. HYBRIDKERNEL DESIGN

The HybridKernel architecture is presented in Figure 1. It constructs from the preemptive kernel and from the novel event-driven extension. In HybridKernel, the preemptive threads are named as *processes* and the protothreads are named as *threads*. The naming convention was chosen to emphasize overhead differences and to remove confusion between the preemptive threads and the protothreads.

3.1. Preemptive kernel

Processes are scheduled by the process scheduler, which is a priority based round-robin scheduler without time-slicing abilities. The highest priority execution ready process is executed until it calls a blocking system call. Every process has at least one thread and the threads are connected to the parent process.

The event handler passes events, which are the communication mechanism of the kernel. Figure 2 illustrates the event handling scheme example. The events are assigned and delivered to the parent process and the thread scheduler is responsible to deliver events to the correct threads.

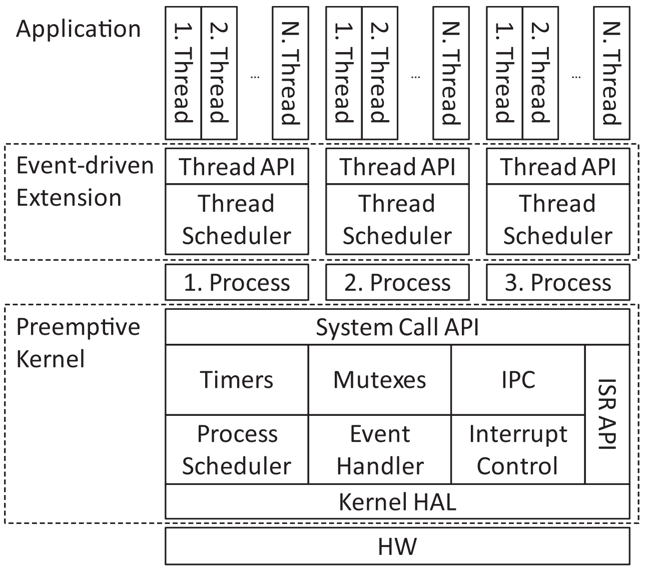


Fig. 1: The Architecture of HybridKernel

The IPC messaging system is provided for interprocess communication [12]. With it processes and threads may deliver message envelopes to each other. The IPC system associates a thread specific message queue to an event. If the receiver tries to receive an IPC message and the message queue is empty, the event is assigned for it. The receiver may then wait until the associated event rises.

Mutual exclusion (mutex) [12] is used for synchronization. The mutex associates a global resource to an event. A process or thread may try to reserve the resource. If the resource is not reserved, the reservation is given to the reserver. If the resource is already reserved, an event is assigned for the reserver. When the reservation holder releases the resource, next event in the reservation list is risen and the event holder is invoked.

The timing interface constructs from high and low resolution timers. The high resolution timer is a microsecond scale timer. It may be used only by one process at a time. To guarantee deadlines to this process, its priority is set to the highest priority. The low resolution timer is a millisecond scale timer and it has no restrictions for amount of using processes or threads. Deadlines of the low resolution timers are not guaranteed and process priorities are not touched.

A Hardware Abstraction Layer (HAL) is used to hide the underlying MCU from the kernel implementation. This ensures fast portability of the kernel.

An Interrupt Service Routine (ISR) API is designed for the interrupt driven device drivers. The device driver may register a callback function and/or an event to an interrupt. When the interrupt occurs, the ISR calls the callback function. The callback function can indicate if the ISR should raise the provided event. If the callback function is not provided, the ISR only raises the provided event.

3.2. Event-driven extension

The thread scheduler is responsible for call every thread inside one process. This is also illustrated in Figure 2. When the application programmer introduces more than one thread to the process, thread scheduler is used. It calls thread entries in the round robin style.

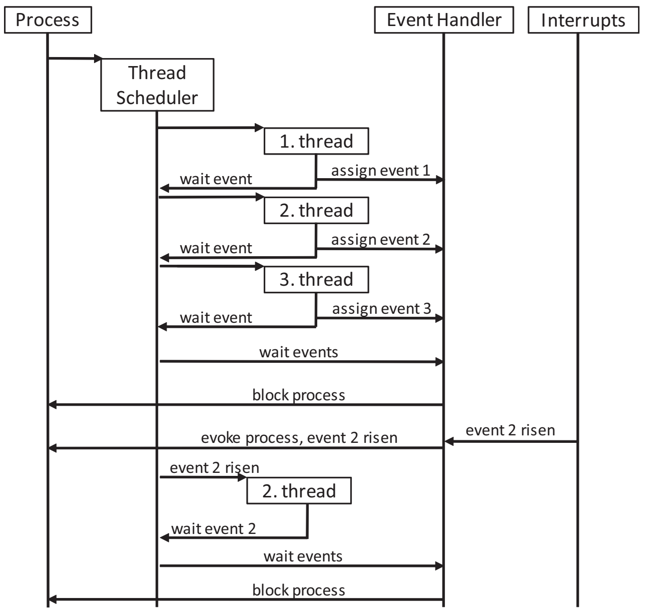


Fig. 2: Event Handling in HybridKernel

Every thread has a state, which indicates if it is blocked, suspended or ready for the execution. When the thread scheduler notices that all the threads are blocked (waiting for an event) or suspended, it starts waiting for an event and blocks the parent process.

The proposed novel event-driven extension does not follow the traditional event-driven approach, since the threads are not necessarily associated to the events. They may be used as event handlers or more like traditional threads. The protothreads fit to this approach perfectly. For example, a nonblocking idle thread may be used to preserve energy efficiency. The idle thread should be in the lowest priority process, so that it is taken to the execution, when all other processes and threads are blocked.

4. IMPLEMENTATION

The prototype applications and HybridKernel were implemented for a prototype WSN hardware platform which consists of a Microchip PIC18F8722 8bit MCU. It has 128KB of program memory and 3938B of data memory. HybridKernel was implemented with C -programming language. Parts of the HAL and the context swap were implemented with assembly.

Process related information is stored to a Process Control Block (PCB). Respectively, thread related information is stored to a Thread Control Block (TCB). The PCB holds a process identifier (PID), a context, a process state, an event queues, and a thread table. The TCB holds a thread entry point, a state, an IPC message queue, a mutex entry, and an alarm entry.

The PCBs are stored to two separate queues depending on the process states. When the process has a state *ready*, it is stored to the ready queue. While the process has state *blocked*, it is stored to a blocked queue.

Processes and threads have a PID. The threads inherit parent processes PID and extend it with their own PID. The kernel uses only the process PID and the thread scheduler uses the thread part. This guarantees that events get assigned to right processes and the thread PID is stored to the event also.

Table 1: HybridKernel system call interface

Process API
os_process_create(**pid**, **options**, **priority**, **code_entry**, **stack**, **thread_table**)
Thread API

os_thread_begin()
os_thread_end()
os_thread_exit()
os_thread_event_wait()
os_thread_yield()

Thread Control API

os_thread_spawn(**pid**, **code_entry**)
os_thread_suspend(**pid**)
os_thread_get_state(**pid**)
os_thread_resume(**pid**)
os_thread_reset(**pid**)
os_thread_kill(**pid**)

Event API

os_event_assing(**event**)
os_event_deassign(**event**)
os_event_wait(**block**)

Timers API

os_get_entry_time()
os_wait_until(**time_stamp**)
os_alarm_set(**time**)
os_alarm_remove()

Mutexes API

os_mutex_reserve(**mutex**, **block**)
os_mutex_release(**mutex**)

IPC API

os_ipc_send(**msg**)
os_ipc_recv(**block**)

Memory control API

os_mem_alloc(**size**)
os_mem_free(**pointer**)

The protothreads were implemented with the same principle as in [5]. The implementation relies on the ANSI C feature, which lets case -statement in a switch -structure to be located anywhere inside the structure. The point of the execution is stored to the so called local continuation variable. With this variable, the protothread can continue execution after a blocking system call.

Table 1 presents the coherent system call API of HybridKernel. It is divided in several APIs to emphasize related system call functions.

Processes may be created with *os_process_create* system call function. It takes a PID, options, priority, process code entry point, stack and thread table as parameters. With the parameter *options* the application programmer may decide whether or not the process has a thread scheduler. If the thread scheduler is not included, the process may be used like a traditional preemptive thread. Stack and thread table allocations are left to the application programmer.

The application programmer may create new threads with *os_thread_spawn* function. A PID and a code entry point are needed to spawn a new thread. If maximum amount of threads is exceeded *os_thread_spawn* returns an error code. With functions *os_thread_suspend*, *os_thread_resume*, *os_thread_reset* and *os_thread_kill* application programmer may control existing threads in the system.

Every thread has to start with *os_thread_begin* and end with *os_thread_end*. A thread may exit permanently with *os_thread_exit*. When thread exits, its resources are freed and a new thread may be spawn to replace it. Thread may voluntarily yield with *os_thread_yield*.

An event may be assigned with *os_event_assign*. If the event is already assigned, an error code is returned. The thread may wait events with *os_thread_event_wait*.

System calls *os_entry_time* and *os_wait_until* are for the high resolution timer. The *os_entry_time* returns current time stamp and the *os_wait_until* returns when the given time stamp has passed. The *os_alarm_set* and *os_alarm_remove* system calls are for the low resolution timer. The *os_alarm_set* takes a millisecond value in which an alarm event is risen. The *os_alarm_remove* removes the currently set alarm.

A resource may be reserved with *os_mutex_reserve* system call and released with *os_mutex_release*. An IPC message may be send and received with *os_ipc_send* and *os_ipc_recv* system calls. A simple memory allocation scheme is provided for dynamic runtime allocations. With *os_mem_alloc* the application programmer may reserve memory block of a desired size. *os_mem_free* can be used to release reserved memory blocks.

The threads may use any system call freely, except the *os_wait_until* and *os_event_wait*. These are blocking system calls for the one threaded processes only. The *os_set_alarm*, the *os_mutex_reserve*, and the *os_ipc_recv* system calls take a parameter that decides whether or not the system call should block the whole process. If they do not block, an event is assigned for the thread and the system call returns a handle to that event.

5. EVALUATION

In this section advantages of the HybridKernel are evaluated. System call API coherency is compared to other hybrid WSN RTOSes. Scalability is evaluated as data memory and program memory consumption. Performance shows timing accuracy and context overhead minimization.

5.1. System call API Coherency

The coherency of the API is achieved as threads and processes may use provided system calls as liked, except the *os_wait_until* and *os_event_wait* system calls. Contiki and TOSTthreads provide distinct library and system call API to support preemptive multithreading. In Contiki, the library is used on top of the event-driven kernel and it uses same event mechanism [10]. However, the event-driven API is not usable crosswise with the preemptive library. In TOSThreads, one thread executes TinyOS [9]. Thus, the TinyOS API cannot be used in other threads and a separate system call API is needed for threads to access TinyOS services. The API of LIMOS is not presented in [11].

5.2. Scalability

The HybridKernel implementation requires 89 bytes of data memory. This is the common figure that is given for kernel memory consumption and it is static. However, when an application introduces tasks, memory consumption increases due to the required PCBs, TCBs, and possible stacks. With the current implementation, one PCB requires 28 bytes and one TCB requires 31 bytes.

The total data memory consumption C can be evaluated with

$$C = 89 + (31 + S) \times \min(T, P) + 28 \times T, \tag{1}$$

where P is the amount of the preemptive processes and T is the amount of threads in the configuration. S is the used stack size for the processes. $\min(T, P)$ is needed due to the fact, that every process need at least one TCB. Invariable 89 comes from the static kernel consumption and invariables 31 and 28 represent the TCB and PCB consumptions.

For example, if the application programmer has 1 kilobyte budget, 5 processes and 6 threads or 3 processes and 16 threads could be used.

With WSN protocols, such as the TUTWSN [2] [13], one high priority process could be dedicated to the Time-Division Multiple Access (TDMA) Medium Access Control (MAC) protocol. One process could handle the routing and management layer with help of two threads and third process could handle the application layer and applications with the remaining threads. TDMA MAC protocols generally require strict timings, since the medium is divided in to time slots. The routing and management protocol layers require CPU time frequently, but they do not need strict timings. The applications traditionally sample sensors with long intervals and do not require frequently CPU time. Thus, this example division is realistic, because the MAC layer has guaranteed deadlines and the routing and management are executed prior to the applications.

Estimations of the data memory consumptions of various WSN RTOSes are gathered in Table 2 for the preceding WSN protocol example. It should be noted that stack size and control blocks varies depending of the used MCU and the compiler. Estimations are based to the given values if possible. However, some TCB and PCB estimations are based on the available source codes. Thus, these estimations are suggestive.

16 threads are used for the preemptive SensorOS [3] and MantisOS [8] kernels. For Contiki, TinyOS, and HybridKernel three preemptive parts and 16 cooperative parts are used. The stack size of 128 B is used. Contiki requires one stack for the event-driven part and thus it has 128 B more stack consumption than others. To avoid confusion between processes and threads, the PCB is used for the preemptive part memory consumption and the TCB is used for the nonpreemptive parts.

The estimated values show the benefits of the hybrid approach. The SensorOS and MantisOS require more data memory and thus they are harder to fully utilize with small 8 bit MCUs. HybridKernel and Contiki have similar memory consumption figures, where as TinyOS approach requires more data memory. As a conclusion, HybridKernel has one of the smallest data memory footprints of the presented kernels and it scales to various configurations.

HybridKernel uses 9038 bytes of the program memory when compiled for the Microchip PIC18F8722 MCU. This is 7 % of the available program memory. Comparison to other RTOSes is not feasible, since the compiler and the target platforms vary. Preemptive WSN RTOS SensorOS is implemented for similar Microchip PIC18F4620 MCU in and it requires 6964 B of the program memory [3]. Thus, the event-driven extension brings about 2 KB of program memory overhead.

Table 2: Data memory footprint comparison of WSN RTOSes based on estimated values

RTOS	Kernel (bytes)	PCB (bytes)	PCB total (bytes)	Stacks (bytes)	TCB (bytes)	TCB total (bytes)	Total (bytes)	Difference (%)
HybridKernel	89	28	84	384	31	496	1053	0 %
SensorOS	115 [3]	17 [3]	272	2048	-	-	2435	131 %
TinyOS	178 [7]	43 [14]	129	384	46 [7]	736	1427	35 %
Contiki	230 [10]	8 [15]	24	476	15 [16]	240	1006	-5 %
MantisOS	144 [8]	10 [8]	160	2048	-	-	2352	123 %

5.3. Performance

The high resolution timer accuracy was tested with an application, where the CPU was heavily populated at every timer invoke by the lower priority processes with several threads. Timing accuracy was tested on a continuous manner, where the highest priority process waited the high resolution timer at constant intervals. This test ensures that the deadline can be guaranteed for the high resolution timer user, such as the TDMA based MAC -protocol. Test was performed with 250 ms, 500 ms and 1 s wait intervals. The results show that accuracy was constant through different intervals and average variance was as low as 2 μs.

The context switching takes 90 μs on average. With HybridKernel there are eventually less context switches than with the preemptive only kernels. For example, if the application requires accurate execution for one thread with 100 ms intervals and between every 100 ms three other threads execute their tasks. When the three threads are ready, the CPU is released for the idle thread. With the traditional preemptive kernel, the context switching overhead would be 0.72 % of the execution time. With HybridKernel the overhead would be 0.18 %, if the timing critical thread is executed in one process and other threads are executed in lower priority process.

The results show that the design of HybridKernel does not affect the timing accuracy of a preemptive kernel. The minimizing of the context switching overhead improves energy efficiency, due to decrease in the kernel processing time.

6. CONCLUSIONS AND FUTURE WORK

This paper shows that using an event-driven extension with a preemptive RTOS kernel is viable. The presented HybridKernel has a small data memory footprint, it preserves time accurately, it is scalable, it improves energy efficiency, and it provides an easy-to-use coherent system call interface. All these are achieved with small program memory overhead and small increase in the responsibility of the application programmer. HybridKernel enables preemptive kernels with small 8 bit microcontroller units. Thus, it enables preemptive kernels with the resource constrained WSN platforms.

In future work, we will study a more dynamic priority based thread scheduler. A dynamic loading of the protothreads over WSN is another aspect of the future work.

7. REFERENCES

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

[2] Mikko Kohvakka, Marko Hännikäinen, and Timo D. Hämäläinen, "Ultra low energy wireless temperature sensor network implementation," in *Proc. 16th Annual IEEE International Symposium on Personal Indoor and Mobile Radio Communications*, Berlin, Germany, Sept. 2005, pp. 801–805.

[3] Mauri Kuorilehto, Timo Alho, Marko Hännikäinen, and Timo D. Hämäläinen, "Sensoros: A new operating system for time critical wsn applications," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Heidelberg, Germany, Aug. 2007, pp. 431–442.

[4] R. Dasgupta, "Anatomy of rtos and analyze the best-fit for small, medium and large footprint embedded devices in wireless sensor network," *Sensor Technologies and Applications, 2008. SENSORCOMM '08. Second International Conference on*, pp. 598–603, Aug. 2008.

[5] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali, "Protothreads: Simplifying event-driven programming of memory-constraine embedded systems," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, Boulder, Colorado, USA, Nov. 2006, pp. 29–42.

[6] Cormac Duffy, Utz Roedig, John Herbert, and Cormac Sreenan, "An experimental comparison of event driven and multi-threaded sensor node operating systems," *Pervasive Computing and Communications Workshops, 2007. PerCom Workshops '07. Fifth Annual IEEE International Conference on*, pp. 267–271, March 2007.

[7] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister, "System architecture directions for networked sensors," *SIGPLAN Not.*, vol. 35, no. 11, pp. 93–104, 2000.

[8] Shah Bhatti, James Carlson, Hui Dai, and et al., "Mantis os: An embedded multithreaded operating system for wireless micro sensor platforms," *Mobile Networks and Applications*, vol. 10, no. 4, pp. 563–579, Aug. 2005.

[9] TinyOS Alliance, "Tinyos 2.1 adding threads and memory protection to tinyos," in *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, New York, NY, USA, 2008, pp. 413–414, ACM.

[10] Adam Dunkels, Björn Grönvall, and Thiemo Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proc. 29th Annual IEEE International Conference on Local Computer Networks*, Tampa, FL, USA, Nov. 2004, pp. 455–462.

[11] Hai ying Zhou and Kun mean Hou, "Limos: A lightweight multi-threading operating system dedicated to wireless sensor networks," *Wireless Communications, Networking and Mobile Computing, 2007. WiCom 2007. International Conference on*, pp. 3051–3054, Sept. 2007.

[12] Abraham Silberschatz and Peter B. Galvin, *Operating system concepts*, Addison-Wesley Publishing Company, 4 edition, 1994.

[13] Jukka Suhonen, Mauri Kuorilehto, Marko Hannikainen, and Timo D. Hamalainen, “Cost-aware dynamic routing protocol for wireless sensor networks - design and prototype experiments,” *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, pp. 1–5, Sept. 2006.

[14] TinyOS code repository, “File: tinyos-2.x/tos/lib/tosthreads/types/thread.h, revision 1.1,” [ONLINE] http ://tinyos.cvs.sourceforge.net/, Apr. 2009.

[15] Contiki code repository, “File: contiki-2.x/core/sys/mt.h, revision 1.6,” [ONLINE] http ://contiki.cvs.sourceforge.net/, Apr. 2009.

[16] Contiki code repository, “File: contiki-2.x/core/sys/process.h, revision 1.16,” [ONLINE] http ://contiki.cvs.sourceforge.net/, Apr. 2009.

PUBLICATION 2

T. Laukkarinen, L. Määttä, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "Design and Implementation of a Firmware Update Protocol for Resource Constrained Wireless Sensor Networks". *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, 2(3), pp. 50–68, 2011. doi:10.4018/978–1–4666–2776–5.ch003

Design and Implementation of a Firmware Update Protocol for Resource Constrained Wireless Sensor Networks

Teemu Laukkarinen, Tampere University of Technology, Finland

Lasse Määttä, Tampere University of Technology, Finland

Jukka Suhonen, Tampere University of Technology, Finland

Timo D. Hämäläinen, Tampere University of Technology, Finland

Marko Hännikäinen, Tampere University of Technology, Finland

ABSTRACT

Resource constrained Wireless Sensor Networks (WSNs) require an automated firmware updating protocol for adding new features or error fixes. Reprogramming nodes manually is often impractical or even impossible. Current update protocols require a large external memory or external WSN transport protocol. This paper presents the design, implementation, and experiments of a Program Image Dissemination Protocol (PIDP) for autonomous WSNs. It is reliable, lightweight and it supports multi-hopping. PIDP does not require external memory, is independent of the WSN implementation, transfers firmware, and reprograms the whole program image. It was implemented on a node platform with an 8-bit microcontroller and a 2.4 GHz radio. Implementation requires 22 bytes of data memory and less than 7 kilobytes of program memory. PIDP updates 178 nodes within 5 hours. One update consumes under 1‰ of the energy of two AA batteries.

Keywords: *Computer Science, Dissemination Protocol, Embedded Systems, Reprogramming, Wireless Sensor Networks*

INTRODUCTION

A Wireless Sensor Network (WSN) consists of autonomous sensor nodes (Akyildiz, Weilian, Sankarasubramaniam, & Cayirci, 2002). The goal of sensor node hardware development is to create tiny battery-powered low-cost disposable nodes. Increasing the performance or memory capacity increases the physical size, energy consumption and manufacturing costs. Thus, nodes are limited in computation, storage, communication and energy resources. These limitations must be addressed when designing and implementing protocols in WSNs.

DOI: 10.4018/jertcs.2011070103

It is not always possible to physically access the nodes in the field once they are deployed. Yet, adding new features, applications and program error fixes necessitates updating the program image that contains the software and protocols running on a node. The solution is a WSN reprogramming protocol, which is used to inject new software into a WSN.

Five general challenges affecting reprogramming in WSNs can be identified (Wang, Zhu, & Cheng, 2006). First, large program images must be transferred reliably through an error prone medium. Thus, the receiver should be able to detect errors and request the corrupted segments again. Second, processing speed and memory capacity in nodes set limits to the time and space complexity of designed protocols. Third, battery powered WSN nodes inherently require the reprogramming protocols to be energy efficient. Fourth, the reprogramming protocol must be scalable enough to handle WSNs that consist of hundreds or thousands of nodes deployed in varying densities. And fifth, the operating system, which is used in nodes, can set limits on the program image format and the reprogramming protocol.

Several protocols (Wang, Zhu, & Cheng, 2006) have been proposed for reprogramming a WSN. A common approach is to equip each node with external memory storage where the new program image is stored. Once the image has been received and verified, a dedicated image transfer program copies the new program image over the old image. This approach allows uninterrupted operation as the new image is transferred in the background. However, the additional memory increases hardware price and takes place on the circuit board, therefore necessitating expensive or energy consuming platforms that prohibit the vision of long term, disposable nodes. Furthermore, many protocols (Hui & Culler, 2004; Levis, Patel, Culler, & Shenker, 2004; Levis & Culler, 2002) support a particular operating system only.

In this paper we present the design, implementation and experimental results of a Program Image Dissemination Protocol (PIDP) for autonomous adhoc multihop WSNs. PIDP consists of firmware version handshakes between nodes, periodic firmware version advertisements and a reliable program image transfer, as shown in Figure 1. Firmware version advertisements are used between neighboring nodes to advertise and compare firmware versions and check for compatibility. The reliable image transfer is used to transfer program images between nodes and to rewrite the program memory. A small bootloader program locates and executes the loaded program image. PIDP is lightweight, energy efficient, reliable and, unlike other reprogramming protocols, does not require external memory for temporary storage of program images. A PIDP update in one part of the WSN does not disturb the whole network, thus, allowing a continuous operation of the non-affected nodes. Furthermore, PIDP is not restricted to a particular operating system or WSN protocol.

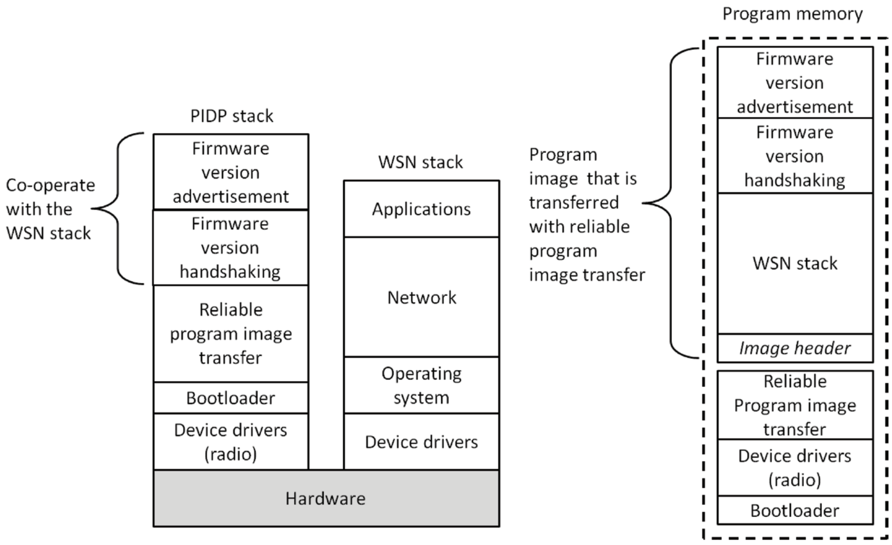
PIDP was evaluated using the TUTWSN prototype (Kuorilehto, Kohvakka, Suhonen, Hämäläinen, Hännikäinen, & Hämäläinen, 2007). TUTWSN is a state of the art adhoc multihop WSN technology for resource-constrained WSNs developed by Department of Computer Systems at Tampere University of Technology. TUTWSN features an energy efficient medium access control (MAC), which uses time-division multiple access (TDMA), a cost-aware routing protocol (Suhonen, Kuorilehto, Hännikäinen, & Hämäläinen, 2006) and multiple custom designed hardware platforms. The operating principle of the MAC layer of TUTWSN is similar to the beacon enabled clustered mode in IEEE 802.15.4 (IEEE Standards Association, 2008). Therefore, a similar implementation can be applied to ZigBee (ZigBee Alliance, 2010).

The paper is organized as follows. First, related work is covered. Second, the design of PIDP is presented. Third, implementation is shown. Fourth, evaluation, performance measurements are given. Finally, the paper is concluded.

RELATED WORK

A number of reprogramming protocols for WSN are built on the TinyOS (Hill, Szewczyk, Woo,

Figure 1. The logical structure and the memory layout of PIDP and the WSN stack. PIDP is a separate protocol stack. Firmware version advertisements and handshaking co-operate with the WSN stack to disseminate version information and to begin reliable program image transfer.


PIDP stack
Co-operate with the WSN stack
Firmware version advertisement
Firmware version handshaking
Reliable program image transfer
Bootloader
Device drivers (radio)
WSN stack
Applications
Network
Operating system
Device drivers
Hardware
Program image that is transferred with reliable program image transfer
Program memory
Firmware version advertisement
Firmware version handshaking
WSN stack
Image header
Reliable Program image transfer
Device drivers (radio)
Bootloader

Hollar, Culler, & Pister, 2000) operating system. TinyOS does not support loadable modules. Thus, a program image must be loaded as a single binary image.

XNP (Crossbow Technologies, 2003) is one of the first reprogramming services for TinyOS and the MICA2 platform. It features a single-hop reprogramming scheme where the program image is sent as unicast to a particular node or broadcasted to a group of nodes. The single-hop nature limits the scalability of XNP and it only serves as an alternative to manual wired reprogramming.

The successor to XNP is Deluge (Hui & Culler, 2004). Deluge is an epidemic multihop protocol that allows nodes to store several different program images in an external EEPROM memory. One of these images can act as the so called Golden image, which is used as a backup image if the main program image is corrupted. The 2.0 version (Hui, 2005) also adds support for resuming incomplete program image downloads and additional program image verification. MOAP (Stathopoulos, Heidemann, Estrin, & SENSING, 2003) is similar to Deluge.

The Maté virtual machine (Levis & Culler, 2002), which is built upon TinyOS, bypasses the lack of loadable modules by presenting a high-level virtual machine instruction set. Maté bytecode programs are smaller than full program images, which lowers the energy cost of disseminating them. The downside is that interpreting the bytecode creates energy overhead. If new software is disseminated only seldom, the energy consumption of the code interpretation is dominant.

Unlike the TinyOS-based approaches, individual applications and services can be loaded individually in the Contiki operating system (Dunkels, Gronvall, & Voigt, 2004; Dunkels, Finne, Eriksson, & Voigt, 2006). Like Maté, this saves energy as only parts of the whole image need to be disseminated. This dynamic loading only applies to the applications, while the operating system and the protocol stack can only be updated with a separate special image transfer program.

The requirement for external memory storage is common to all these reprogramming protocols, as they use transport layer dissemination protocols to transfer program images. These dissemination protocols are stored within the main program image, which cannot be overwritten as long as it is being executed. This can cause problems e.g. in (Langendoen, Baggio, & Visser, 2006), where unreliable MAC protocol made Deluge useless. As a result, nodes were updated by hand on the deployment site.

An approach to updating based on the differences between the old and the new program image is presented in (Mukhtar, Kim, Kim, & Joo, 2009). The old and new images are analyzed and a model to modify the old one is created. The model and completely new parts are disseminated with any dissemination protocol. Similar approach is presented in (Reijers & Langendoen, 2003), but this is a processor specific solution. These approaches do not efficiently update the image when it is significantly different from the old image. However, these do not require external flash, but they do require a reliable transport protocol layer. Reliable transport protocols for code dissemination have been presented in Stathopoulos, Heidemann, Estrin, and SENSING (2003) and Miller and Poellabauer (2008).

As opposed to Contiki, Maté or difference models, PIDP transfers complete program images. In our experience, the ability to update individual applications is seldom needed as programming error fixes and new features often affect multiple modules of the program image. In addition, loading individual applications requires either support from the operating system or a separate mechanism for handling runtime relocation of modules. PIDP requires no such support and is operating system independent. PIDP does not require an external reliable transport protocol and it can be used to update completely different image to the network.

PIDP DESIGN

PIDP design consists of firmware version handshaking, periodic firmware version advertisements, and reliable image transfer. Figure 2 presents the PIDP design in action. Following sections present the design in detail. PIDP minimizes communication and memory overhead, therefore allowing very resource-constrained implementations. Also, PIDP design includes new firmware injection, security, operating system support, and support for heterogeneous networks.

Firmware Version Handshaking

Program image transfer begins automatically when a node detects that one of its neighbors has a new version of a compatible program, as shown in Figure 2 between the nodes A and B. The node with a lower version number sends a *firmware request* and the other node responds with a *firmware confirmation* at the WSN protocol level. After this, the nodes jump to the reliable image transfer of the PIDP, which is independent of the WSN stack.

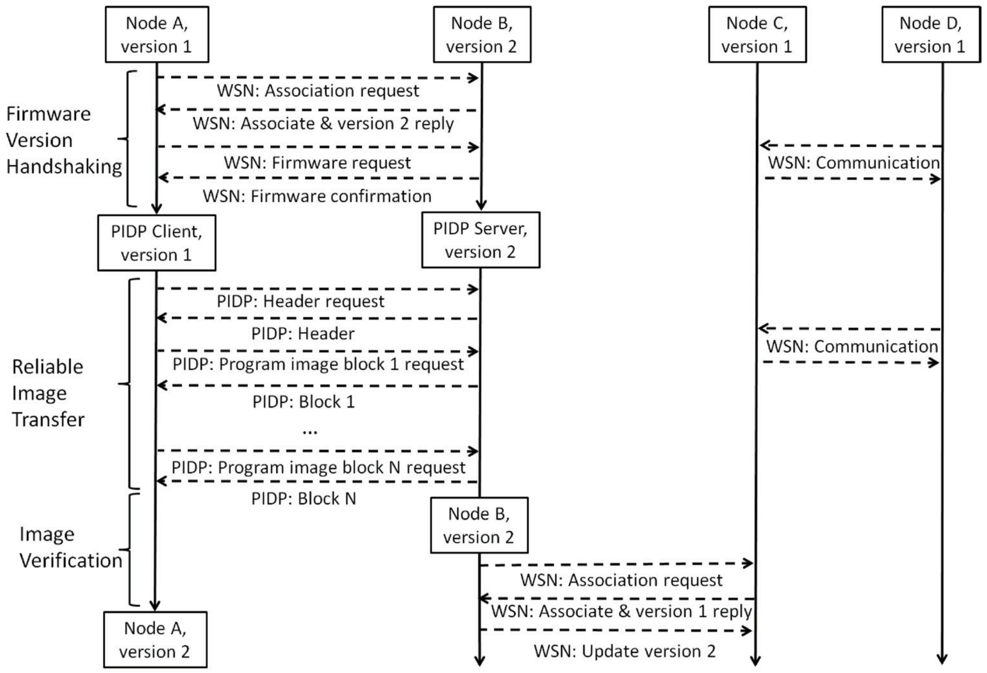
PIDP assumes that a node performs a handshake with its neighbors after powering up to find new routes. This is the case in most of the sender decided WSN protocols, such as ZigBee. Version information is exchanged in PIDP when a node exchanges routing information or synchronizes with its neighbors, which adds a small overhead. Either node participating in the handshaking can start the update operation. Both nodes reboot after the update and perform handshaking with their neighbors. This guarantees that program images will propagate epidemically in the WSN.

It is important to note that version information is exchanged on a hop-by-hop basis between neighbors without flooding the version information further into the network. If a network contains multiple nodes with incompatible program images, it may limit the propagation of program images.

Periodic Firmware Version Advertisements

Nodes periodically advertise their program image version on an advertisement channel, as shown in Figure 3. After each advertisement the source listens for a reply. The parameter T_a

Figure 2. Node B has a newer version. Node A and Node B execute firmware handshaking on WSN association. Then they start program image update and move to the PIDP reliable image transfer. Meanwhile Node C and D continue normal WSN operation. Eventually, Node A is updated and nodes reboot. Node B associates with Node C, starts the update, and continues disseminating image further.

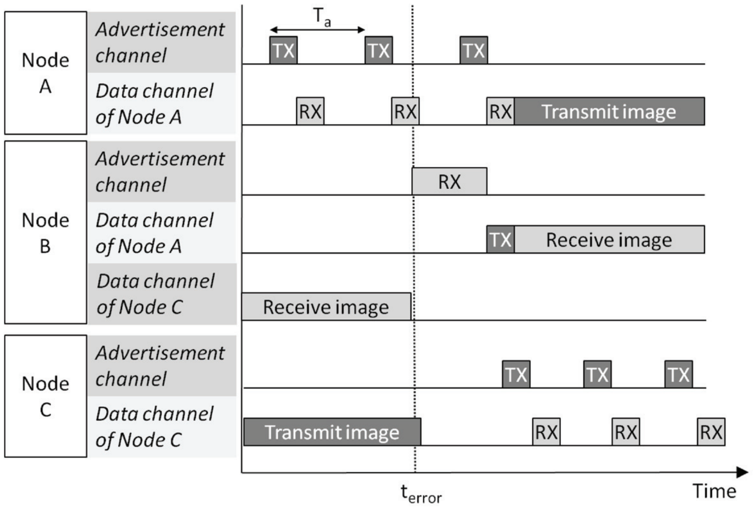


adjusts the interval between these periodic advertisements. The main purpose of the periodic advertisements is to act as a failsafe. If a node encounters a problem while reprogramming or the image transfer is disrupted, the node may listen for the periodic advertisements to find a new source for image transfer as shown in Figure 3. In addition, periodic advertisements allow nodes to perform image acquisition even if their protocol stacks might otherwise be incompatible. As periodic advertisements are only transmitted seldom and nodes do not listen for them during normal operation, they use very little energy. WSNs that use synchronized MAC protocols, such as IEEE 802.15.4, can embed the version information in synchronization beacons, which nodes transmit periodically.

Reliable Image Transfer

Information about the program image is stored in a header, which consist of hardware identifier, firmware version, message authentication code, and valid and dissemination bits. Program images are identified by a combination of the hardware platform identification number and the firmware version number. Platform identification numbers are used to limit the transfer of program images between incompatible sensor nodes. Furthermore, the header contains a valid bit that indicates whether or not the program image has been successfully validated and can be safely executed. The dissemination bit decides if the node will disseminate the image to the network.

Figure 3. Example of periodic advertisements and their functioning as a fail-safe. Node A transmits advertisements with interval T_a. Node B is updating its firmware with Node C at the data channel of Node C. Node B encounters a problem at t_{error}, scans the advertisement channel and begins a new transmission with Node A using the data channel of Node A.



Unlike other reprogramming proposals, the image transfer in PIDP operates independently of the main WSN stack. This allows the image transfer protocol to achieve a better energy-efficiency, as minimizing the number of protocols layers used in the transfer also minimizes the amount of overhead in the transmission of program images. Furthermore, simple independent stack can be tested thoroughly and possible problems of unreliable transfer protocols cannot prevent program image update. As the WSN protocol stack is part of the updated image, the possible problems on WSN protocols can be fixed with PIDP.

The image transfer protocol follows the general client-server architecture as seen in Figure 2. The receiver of the program image acts as a PIDP client, while the sender acts as a PIDP server. Communication between a PIDP client and a PIDP server is performed at a channel selected by the PIDP server, which is transmitted within the firmware advertisements. PIDP servers may choose to use a single network-wide dedicated channel for the image transfers or they may use an appropriate channel selection algorithm to choose channels that are not being used. Choosing different channels is preferred, as this lowers the chance of collisions with nearby image transfers.

The PIDP client begins by requesting the header of the program image as presented in Figure 2. Once the header is received the PIDP client marks the current header invalid and requests the contents of the program image in blocks. After each block the PIDP client immediately writes the data to the program memory, thus invalidating the previous program image. After the whole image is received the PIDP client validates the program image by calculating the message authentication code and comparing it to the one in the header. If the validation calculation is correct, the header is marked valid. Otherwise, the header remains marked invalid.

After the transfer the PIDP client and the PIDP server reboot and return to the normal WSN operation. The PIDP client uses the PIDP bootloader program to check that the header is valid and begins executing code from the beginning of the program image. If the header is not valid then the PIDP client begins to scan the advertisement channel for firmware advertisements and re-executes the image transfer.

Program Image Injection

Three alternative ways exist for new program image injection with PIDP. First, a new node with a new image may be brought to the coverage area of the WSN. The new image is then disseminated to the network automatically by PIDP. Second, a *PIDP cloner device* may be used to transfer program image to one node in the network, which then starts advertising the new version. Third, the new image can be uploaded to a server, which delivers it to the gateways. The gateways advertise the new image to the network and PIDP will first update the nearest nodes using the gateways as relays.

The PIDP cloner device is a specially programmed node that does not act as a part of the WSN. It only advertises the new image on a special cloning channel. The nodes do not normally listen for this channel. When the node is rebooted while a button is pressed, the node will listen for the cloning channel for a period. If there is a PIDP cloner device nearby and the hardware platform identifier match, the node will start the image transfer. If the dissemination bit is set, the newly programmed node will then start disseminating the image further.

The server injection is presented in Figure 4. The image is first compiled and then modified with a script to a XML file, and the XML file is finally uploaded to the database. The server indicates to the gateways that there is a new image to advertise. When a node notices the new image from the advertisements, the gateway requests the new image piece by piece from the server and relays it to the node.

Security

Three major security questions concern program image updating (Deng, Han, & Mishra, 2006). First, the new image must be from a reliable source. Second, the new image must be valid. Third, the image must be transferred securely to preserve intellectual property. PIDP accepts only images with correct message authentication code. This ensures that unknown source cannot inject a new image to the network and hijack the network. The message authentication code is calculated with a one-way function that uses a secret key, the program image, and a magic number as parameters. As the program image is used in calculation of the message authentication code, the image validity is secured at the same time. The secret key can be used to encrypt and decrypt the program image packets with AES algorithm after the handshaking to prevent stealing the program image with sniffing.

Operating System Support

WSN operating systems have two approaches for updating. The whole image including the application and the operating system are disseminated (TinyOS and Deluge), or only the applications are disseminated to the network (Contiki). PIDP supports both ways as presented in Figure 5. The operating system can be a part of the whole program image as in Figure 5a. This is similar to Deluge and TinyOS. The operating system can be left out of the program image, as in Figure 5b, but the applications are treated as one image. Injecting one new application requires re-injecting all the existing applications.

PIDP allows as many image version headers as there are room in the version advertisement packet. Thus, program image can be split in several parts as presented in Figure 5c. These parts can be separately updated. The selected program image part is indicated in the handshaking between the PIDP client and server. Then, the PIDP will update only the selected part. This can be used to inject new applications to the network without injecting the remaining ones again and the operating system can be updated separately. However, each image requires a new header. The header overhead would increase and the amount of applications would be limited. Furthermore, the applications should always fit inside a certain space and some applications would waste the program memory. Every program image requires known entry functions, which will add some complexity in

Figure 4. Program image injection starts with compilation of the image from a source code to a hex file, then formatting it to an xml file, and uploading it to the database. The server will retrieve image information and relay it to the gateways. The gateways will advertise the image to the WSN and relay the image, when a node requests the new image.

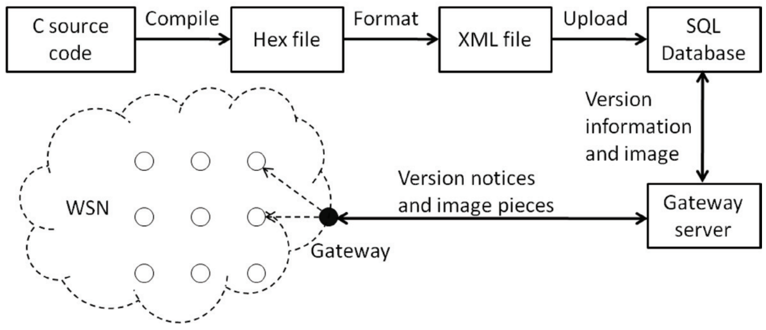
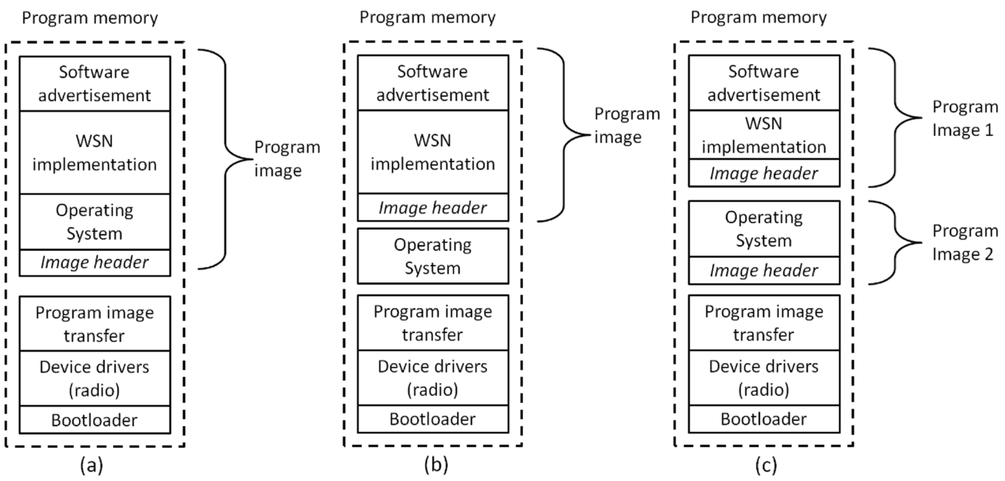


Figure 5. a) A typical use of PIDP, where operating system and WSN stack form the program image. b) If the operating system is reliable and will not require new features, it can be left out of the program image. c) With small modifications, PIDP can update the program image in two or more parts. This allows granular updating, but image headers increase overhead



the development. Multiple program image support is not currently incorporated to the PIDP design nor implemented. It will be implemented in future work. Novel solutions are needed for solving the problems.

PIDP does not restrict the operating system from using its own protocols to update applications. For example, PIDP can update Contiki and Contiki can use its own protocols to handle applications. The image validation should then only cover the area, which is not modified by the program code dissemination of Contiki.

Heterogeneous Node Support

WSNs are seldom homogenous; nodes have different sensors, different roles, and different applications. PIDP separates heterogeneity only between hardware devices. If the hardware is not same, the image transfer is not started. To overcome this limitation, we have developed an auto-configurator. The node is configured during the building process to its configuration: the connected sensors, desired roles and required applications of the node are set to the EEPROM of the node. A program image is used, which contains all the necessary code for these configurable parts. Node selects used role and applications at the startup according to the configuration. This allows us to use one single program image for the whole WSN with various node configurations.

IMPLEMENTATION

PIDP and TUTWSN protocol stack are implemented using the C programming language and the Microchip MPLAB C compiler (Microchip Technology, 2009).

TUTWSN Protocol Stack

The TUTWSN MAC protocol forms a clustered tree topology (Kuorilehto, Kohvakka, Suhonen, Hämäläinen, Hännikäinen, & Hämäläinen, 2007). Each cluster contains a cluster head, a *headnode*, and several cluster members. Cluster members can be leaf nodes or headnodes of other clusters forming a tree of clusters. Each cluster within interference range operates on a separate *cluster channel* that is used for intra-cluster communications. Nodes share a common *network channel* that is used by the headnodes to advertise their clusters. Nodes scan the network channel at least once every hour to find new clusters. In addition, network scans occur when nodes lose their route to the network gateway.

The headnodes maintain a data exchange schedule. Time is divided into fixed length access cycles. Each access cycle begins with a superframe, which contains slots for data transfers at the cluster channel, and ends in an idle time. The length of the access cycle is set to two seconds. Cluster advertisements are sent on the network channel in the beginning of each superframe. Three additional cluster advertisements are also sent during the idle time with approximately 500 millisecond intervals between them.

A TUTWSN sensor node includes an 8 bit Microchip PIC18LF8722 microcontroller (Microchip Technology 2008) with 128 kilobytes of program memory and 3936 bytes of data memory. The microcontroller has an internal 1024 byte EEPROM memory. A Nordic Semiconductors nRF24L01 (Nordic Semiconductors, 2007) is used as the radio, which has a payload size of 32 bytes and a configured transmission rate of 1 megabit per second. The radio does support carrier sensing. It operates on the 2.4 gigahertz band and offers 126 channels and transmission powers of -18 dBm...0 dBm. TUTWSN node has a simple user interface, which consists of a push button and two light emitting diodes. TUTWSN sensor nodes can be equipped with multiple sensors, such as accelerometers, temperature sensors, and humidity sensors. Two 1.5 volt LR6-sized batteries are used as the power source. A TUTWSN sensor node circuit board is shown in Figure 8.

PIDP Implementation

Firmware version handshaking was embedded to the MAC layer of TUTWSN. Thus, nodes exchange version information when they perform association with each other. This allows rapid firmware dissemination within a TUTWSN cluster tree.

The periodic firmware advertisements are transmitted in the TUTWSN network channel, which allows nodes to receive advertisements while they are performing normal neighbor discovery. An advertisement is sent on each access cycle during the idle time. Thus, the interval T_a between the advertisements matches the length of the access cycle. In addition, advertisements on the TUTWSN network channel allow the program image to propagate between different cluster trees, but this method of dissemination is limited by the low frequency of network scans. The cluster channel is used for program image transfer to minimize collisions between concurrent program image transfers.

The bootloader and the program image transfer protocol are stored in a reserved segment in the beginning of the program memory. They are followed with the program image header and the main program image. The message authentication codes are implemented by using a modified 4 byte RC4 code similar to the code described in (Zhang, Yu, Huang, & Yang, 2008).

The program image transfer protocol uses a packet size of 32 bytes. Each packet has a 6 byte header followed by a payload with a length of 26 bytes.

Reliable image transfer is located on a memory section that cannot be updated with PIDP. It includes only the necessary modules to perform the program image transfer and the program memory rewrite. Modules are a radio driver, PIDP server, PIDP client, program memory writer, program image verification, and bootloader. This memory section has to be kept as small as possible since it reduces amount of available memory for the WSN implementation.

The reliable image transfer and the main program are never executed concurrently. Thus, the image transfer can utilize data memory segments that are normally reserved for the main program. Overlaying the data memory significantly reduces data memory requirements of the image transfer protocol. Despite the overlaying, a small amount of dedicated memory is needed for passing version information between the main program and the image transfer.

EVALUATION

Evaluation of PIDP was performed by analyzing the memory consumption, propagation speed and energy consumption impact.

Memory Consumption

Memory consumption was analyzed from the compiled program images for the TUTWSN platform.

From the results in Table 1 we can see that the memory consumption of the PIDP protocol is split in two parts. The first part contains the image transfer while the second part is stored within the program image and contains the necessary support for accessing the image transfer protocol and the implementation of the firmware advertisement scheme.

Although PIDP requires 815 bytes of data memory in total, the absolute increase in the data memory requirements stays at 22 bytes. The image transfer overlays data memory with the WSN stack.

Propagation Time

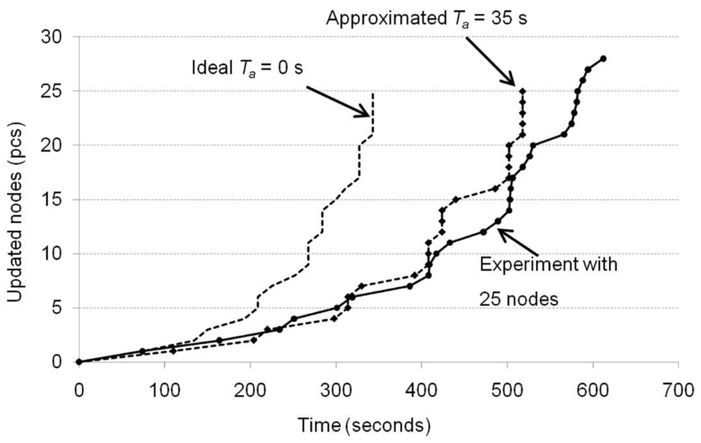
In order to give a reference point for the measured propagation times, the program image transmission and verification times between two nodes were first measured. Transferring a 123 kilobyte image between two nodes in optimal conditions was 51 seconds on average, thus achieving a transfer rate of 2.4 kilobytes per second. The program image verification time was a constant 23 seconds. Thus, the minimum time for updating a sensor node with this particular program image was 74 seconds.

Program image propagation experiments were performed in a typical office environment with various interference sources such as several WLAN routers operating on the same frequency band. The first experiment included one gateway and 25 sensor nodes. The nodes were placed on a table in one group. Size of the table was less than one square meter. The purpose of this experiment was to see how PIDP performed in a situation where every node had multiple neighbors in close proximity and the amount of network activity was high. Update speed is presented in Figure 6. PIDP successfully reprogrammed the nodes in 12 minutes. 8 concurrent image transfers were observed during this period at the time of 500 and 600 seconds from the start. 28 updates were performed, which indicates that 3 updates failed and 3 nodes had to be updated again. Reason for these failures is unknown. T_a was approximated based on the measurements. It took 35

Table 1. Memory consumption of TUTWSN with PIDP in bytes. ©2010 IEEE. Used with permission.

Component	Program memory (B)	Data memory (B)
Reliable image transfer	3578	793 (overlayed)
Version handshaking	1386	15
Firmware advertisements	1425	7
Total	6389	22 + (793)

Figure 6. The updating speed graph of PIDP on the 25 node experiment. The extra three updates were result of failed updates, which caused re-update. Ideal T_a presents how the network would be updated, if a node could start updating another one immediately after receiving the new image. Approximated T_a of the experiment indicates that one node disseminates 35 seconds after the update.

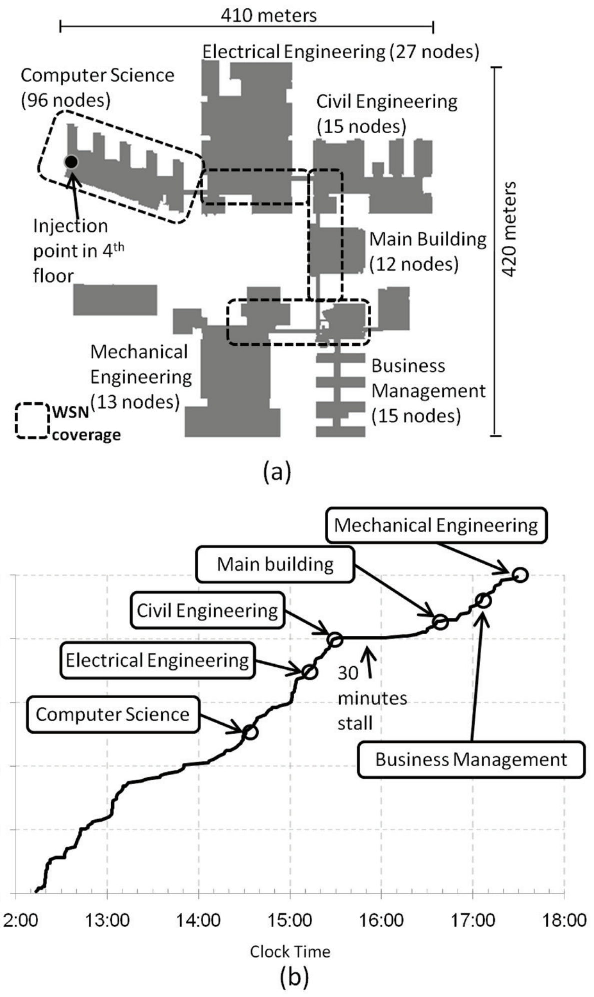


seconds from a node to be capable to disseminate received image.

For the second experiment, the performance of PIDP was measured using the Tampere University of Technology campus WSN. This campus network has 178 sensor nodes and 13 gateways distributed in six buildings around the university campus. Figure 7a presents the campus and the coverage area of the campus WSN. The Computer Science building has sensor nodes in four floors while the others have nodes in only one floor. Distance between nodes ranges from 5 meters to 20 meters. The campus WSN is used as an application platform for students to implement their own applications on a WSN course. Measurement data of the campus WSN is provided for property maintenance. Figure 8 presents a humidity, luminance, and temperature measurement node at the campus of the Civil Engineering building. In addition, carbon dioxide and passive infrared based human activity are measured in the campus WSN. Students attending to the course may carry a node with them, which is tracked by the campus WSN.

The new program image was injected into the WSN by updating a single node on the 4th floor of the Computer Science building. Each node sent a report after a successful update procedure.

Figure 7. a) Tampere University of Technology campus, the coverage of the campus WSN, and the new program image injection point of the experiment in computer science building. b) A graph presenting the TUT Campus WSN experiment progress in percentage of updated nodes. The dissemination was stalled, because two neighboring nodes belonged to different clusters and had good routes to different gateways. Therefore, they did not associate until one hour periodic scan was due. Labels indicate the time when the update was completed for that building.



The results of the second experiment show that PIDP successfully propagated the program image through the campus WSN in five hours, as shown in Figure 7b. A delay was experienced between a pair of nodes located between the Civil Engineering building and the Main building. Once the image had spread to the Main building, it continued to propagate to the rest of the WSN. As a result, program image had to travel over 50 hops to achieve the last node in the Mechanical Engineering building (700 meters long path and average hop distance of 12.5 meters equals 56 hops).

The dissemination speed during the second test was mostly limited by the long interval between network scans. The new image propagated quickly within individual clusters e.g. inside one building, but spread slowly from one cluster to another. As most of the nodes had good routes to the nearest network gateway, they had no reason to perform additional network scans. This limitation can be avoided by disseminating the program image through the network gateways.

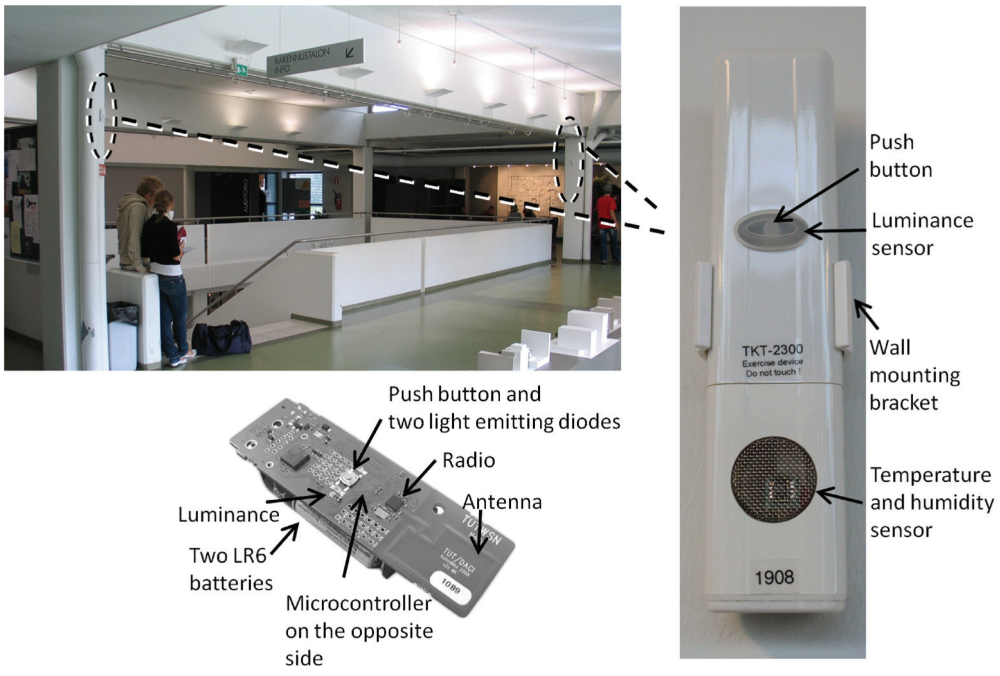
The transmission time between nodes varied from one minute to four minutes. This was caused by the differences in link reliability between different nodes. Due to the hardware restrictions, PIDP client chooses first received advertiser to be a PIDP server without considering link quality. This can lead to a situation where image transfer has to be attempted several times between nodes that are too far apart or suffer from low link reliability due to interference.

Energy Consumption Impact

PIDP energy consumption impact was measured on the TUTWSN hardware platform for the PIDP client and the PIDP server. Measurements were conducted with a stable power source and series resistor of known value. The voltage over the series resistor was measured with an oscilloscope to determine drawn root-mean-square current. Duration and current of the image transfer and verification operations were measured. Slight differences to the measurements in propagation time section are result of different updated image. Figure 9a and Figure 9b present screen captures of the oscilloscope, where the series resistor voltage is drawn over the time.

The PIDP client sends requests to the PIDP server, which are the narrow spikes in Figure 9a. Then the PIDP client listens for the image packets, which are the wide spikes in the capture. One packet has 26 B of payload and the program memory writing has to be done in 64 B blocks.

Figure 8. A TUTWSN node platform, TUTWSN node in an enclosure, and two nodes installed to the Civil Engineering building of Tampere University of Technology campus in the TUT campus WSN


Push button and two light emitting diodes
Radio
Antenna
Luminance
Two LR6 batteries
Microcontroller on the opposite side
Push button
Luminance sensor
Wall mounting bracket
TKT-2300
Temperature and humidity sensor
1908

Therefore, two to three packets are received before writing. The success of writing a block of the program image is always verified immediately. The whole image is verified after the image has been completely transferred. The PIDP server listens for requests of the PIDP client (wide spikes in Figure 9b), prepares a packet, writes it to the radio and sends it to the PIDP client (narrow spikes).

The TUTWSN platform was run on 4 MHz clock frequency and 2.5 V supply voltage. 4 MHz is the highest possible clock frequency with the specified supply voltage. The highest possible clock frequency is used to achieve the fastest possible dissemination time and the shortest affection time to the normal operation. The radio used a random channel from the 2.4 GHz ISM band and the highest possible transmission power of 0 dBm. The MCU is continuously active during the update operation. On the PIDP server, the radio is active for 35 second, which is 67.3% of *the image transfer time* and it is receiving 93.5% of that time. On the PIDP client, the radio is active for 17 seconds, which is 32.7% of *the image transfer time* and it is receiving 90% of that time. These values were obtained from the oscilloscope.

Energy consumption results of the PIDP client are presented in Table 2. Typical lithium AA batteries have approximately 20000 J of usable energy. Thus, one update consumes under 0.1‰ of the available energy of the PIDP client. If the expected node lifetime is two years, updating a node once a day would reduce lifetime approximately 10%. Impact is irrelevant on moderate amounts of updating. However, PIDP is not suitable for continuous application dissemination. This is due to the whole program image updating. If the image is sliced to smaller pieces as presented in Operating System Support section, the energy consumption impact will be reduced.

PIDP server energy consumption is presented in Table 3. The image transfer consumes more energy with the PIDP server, since it has to listen for longer periods. However, the PIDP server does not need to write or verify the image after the transfer and the total amount of consumed energy is similar to the PIDP client. In normal operation, one node acts once as a

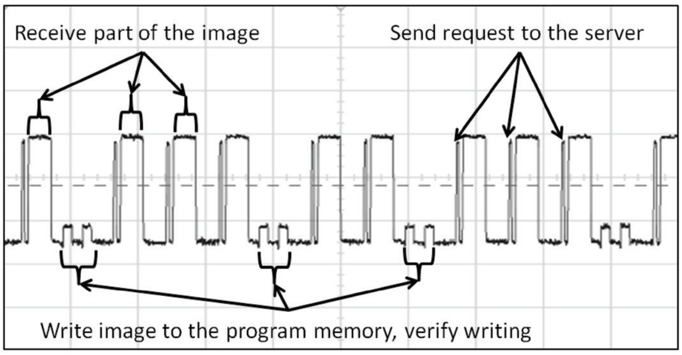
Table 2. Energy consumption measured of the PIDP client. Radio RX denotes for radio listening and receiving. Radio TX denotes for radio transmitting. MCU energy consumption during the image transfer includes the energy consumed in the program memory writing.

Operation	**Time consumed (s)**	**MCU (mJ)**	**Radio RX (mJ)**	**Radio TX (mJ)**	**Total (mJ)**
Image transfer	49.62	711.99	436.14	48.46	1196.59
Image verification	24.44	272.44	0	0	272.44
Total:	74.06	984.43	436.14	48.46	1469.03

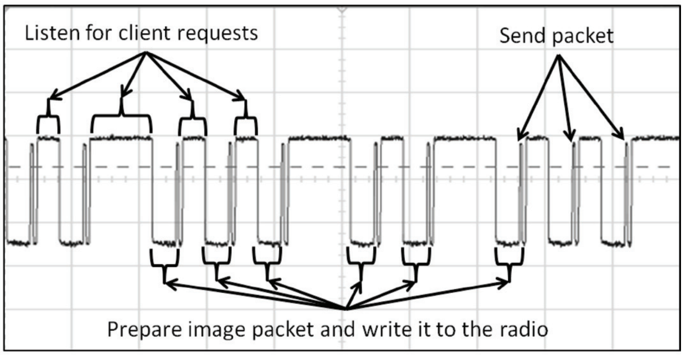
Table 3. Energy consumption measured of the PIDP server. Radio RX denotes for radio listening and receiving. Radio TX denotes for radio transmitting .

Operation	**Time consumed (s)**	**MCU (mJ)**	**Radio RX (mJ)**	**Radio TX (mJ)**	**Total (mJ)**
Image transfer	49.62	553.03	932.92	64.85	1550.80
Image verification	0	0	0	0	0
Total:	49.62	553.03	932.92	64.85	1550.80

Figure 9. Screen captures of the oscilloscope drawing the series resistor voltage during the energy consumption measurements for the PIDP client and server. The scale is horizontally 10 ms/div and vertically 200 mV/div. a) The PIDP client sends a request packet and listens for the image packet. 2-3 image packets are received before writing to the program memory. b) The PIDP server listens for requests of the PIDP client, prepares an image packet and sends it. Note: a) is not in synchronization with b).



(a)



(b)

PIDP client and zero to multiple times as a PIDP server. Therefore, it is difficult to determine actual energy consumption impact of server duty in a network. In our experiments, one node acted as a PIDP server zero to three times. Thus, energy consumption impact varies between 1500 mJ – 6000 mJ, which is under 1‰ of the available energy.

PIDP is an energy efficient method to disseminate new program image to the network. The energy consumption impact increases significantly only if the network is updated often, e.g. once a day. Furthermore, the energy consumption impact is divided evenly across the whole network, since the image disseminates one hop a time and most of the nodes act once as the PIDP client and similar amounts as the PIDP server.

The energy efficiency could be improved by decreasing the radio transmission power or the radio listening time. If the radio transmission power is decreased that the total transmission energy consumption is half of the current consumption, it would reduce energy consumption 2% in total. Thus, the transmission power is not significant energy consumer. In optimum case, radio listening time would be the same as the transmission time. Therefore, reducing the

Table 4. Feature comparison of known WSN reprogramming approaches

Protocol	Requires OS	Supports multiple OS	Requires Ext. Flash	Requires transport protocol	Update Scope
PIDP	No	Yes	No	No	Whole program image or parts
Deluge	Yes / TinyOS	No	Yes	Yes	Whole program image
Contiki	Yes	No	No	Yes	Application dissemination
Maté	Yes / TinyOS	No	No	Yes	Application dissemination
MOAP	Yes / TinyOS	No	Yes	Yes	Whole program image

radio listening time would reduce maximum of 26% of the PIDP client and 56% of the PIDP server energy consumption. Reducing the radio listening time is one major task in future work.

COMPARISON

Feature comparison of known multihop reprogramming methods for WSNs is presented in Table 4. PIDP manages to function without operating system. Also, it can function with any OS and update the OS as well. PIDP does not require external flash or reliable transport protocol, since it has own transport protocol. Finally, PIDP usually updates the whole program image, but it can be modified to update it in parts.

CONCLUSION

This paper presents a lightweight, reliable and energy efficient program image dissemination protocol for WSNs. Unlike other dissemination protocols, PIDP does not require external memory storage, is independent of the WSN stack, offers a low overhead protocol for transferring program images, and can reprogram the whole WSN stack. PIDP is implemented using low-power WSN prototype nodes and tested in actual real-world conditions. The experimental results show that PIDP can reprogram 178 nodes in 5 hours and requires less than 7 kilobytes of ROM and 22 bytes of RAM and that it is possible to create a dissemination protocol that does not require external memory and yet achieves the epidemic dissemination capabilities of traditional dissemination protocols with low energy consumption.

Future work on PIDP will include new methods for inter-cluster advertisements, reliability improvements, energy consumption minimizing, and use of it for application dissemination with operating systems.

The new inter-cluster advertisement methods will speed up the propagation of the program image. This would remove stalls as seen in Figure 7b, where network was partitioned and both partitions considered their network situation satisfactory. In addition to multiple injection points, this can be solved with application level software advertisements, where nodes are informed in the application level that there might be newer program image available. Then the nodes could seek more eagerly for the new image.

To improve reliability, the PIDP protocol must select the transfer channel from non-interfering channels. Also, the PIDP client should start the image transfer with the best possible neighbor. These are difficult tasks to do and require novel designs and implementations to fit the PIDP design.

Energy consumption can be reduced significantly by reducing the radio listening time. This requires strictly synchronized protocol. The PIDP client and server could negotiate a timetable in every transmission for the next packet. This introduces research problems of what to do after unsuccessful transmissions and how to fit such a complex protocol on a restricted space.

For operating systems support, we will implement the two part program image dissemination to PIDP. SensorOS (Kuorilehto, Alho, Hännikäinen, & Hämäläinen, 2007) is used as operating system and WSN API (Juntunen, Kuorilehto, Kohvakka, Kaseva, Hännikäinen, & Hämäläinen, 2006) is used as an application layer. Also, we will experiment with Contiki to see, how the problems of dynamic application loading can be overcome with PIDP.

ACKNOWLEDGMENT

This paper is an updated and revised version of Määttä, Suhonen, Laukkarinen, Hämäläinen, and Hännikäinen, (2010), "Program Image Dissemination Protocol for Low-energy Multihop Wireless Sensor Networks", in Proceedings of the 2010 International Symposium on System on Chip (SoC). IEEE.

REFERENCES

Akyildiz, I., Weilian, S., Sankarasubramaniam, Y., & Cayirci, E. (2002). A survey on sensor networks. *IEEE Communications Magazine*, *40*(8), 102–114. doi:10.1109/MCOM.2002.1024422

Alliance, Z. (2010). *ZigBee specification*. Retrieved from http://www.zigbee.org/Standards/ZigBeeSmartEnergy/Specification.aspx

Crossbow Technologies. (2003). *Mote in-network programming user reference*. Retrieved from http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf

Deng, J., Han, R., & Mishra, S. (2006). Secure code distribution in dynamically programmable wireless sensor networks. In *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks* (pp. 292-300).

Dunkels, A., Finne, N., Eriksson, J., & Voigt, T. (2006). Run-time dynamic linking for reprogramming wireless sensor networks. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems* (pp. 15-28).

Dunkels, A., Gronvall, B., & Voigt, T. (2004). Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks* (pp. 455-462).

Hill, J. W., Szewczyk, R., Woo, A., Hollar, S., Culler, D., & Pister, K. (2000). System architecture directions for networked sensors. *SIGPLAN Notes*, *35*(11), 93–104. doi:10.1145/356989.356998

Hui, J. W. (2005). *Deluge 2.0 - TinyOS network programming*. Retrieved from http://www.cs.berkeley.edu/~jwhui/deluge/deluge-manual.pdf

Hui, J. W., & Culler, D. (2004). 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* (pp. 81-94).

IEEE Standards Association. (2008). *Part 15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for low-rate wireless personal area networks (WPANs)*. Retrieved from http://standards.ieee.org/getieee802/download/802.15.4a-2007.pdf

Juntunen, J., Kuorilehto, M., Kohvakka, M., Kaseva, V., Hännikäinen, M., & Hämäläinen, T. (2006). WSN API: Application programming interface for wireless sensor networks. In *Proceedings of the IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications* (pp. 1-5).

Kulkarni, S., & Wang, L. (2005). MNP: Multihop network reprogramming service for sensor networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems* (pp. 7-16).

Kuorilehto, M., Alho, T., Hännikäinen, M., & Hämäläinen, T. D. (2007). SensorOS: A new operating system for time critical WSN applications. In *Proceedings of the 7th International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (pp. 431-442).

Kuorilehto, M., Kohvakka, M., Suhonen, J., Hämäläinen, P., Hännikäinen, M., & Hämäläinen, T. D. (2007). *Ultra-low energy wireless sensor networks in practice: Theory, realization and deployment*. New York, NY: John Wiley & Sons. doi:10.1002/9780470516805

Langendoen, K., Baggio, A., & Visser, O. (2006). Murphy loves potatoes: Experiences from a pilot sensor network deployment in precision agriculture. In *Proceedings of the 20th International Parallel and Distributed Processing Symposium* (p. 8).

Levis, P., & Culler, D. (2002). Maté: A tiny virtual machine for sensor networks. *SIGOPS Operating Systems Review*, *36*(5), 85–95. doi:10.1145/635508.605407

Levis, P., Patel, N., Culler, D., & Shenker, S. (2004). Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proceedings of the 1st Conference on Networked Systems Design and Implementation* (p. 2).

Microchip Technology. (2008). *PIC18F8722 product page*. Retrieved from http://www.microchip.com/

Microchip Technology. (2009). *MPLAB C compiler for PIC18 MCUs*. Retrieved from http://www.microchip.com/

Miller, C., & Poellabauer, C. (2008). PALER: A reliable transport protocol for code distribution in large sensor networks. In *Proceedings of the 5th Annual IEEE Communications Society Conference on Sensor, Mesh and Ad Hoc Communications and Network* (pp. 206-214).

Mukhtar, H., Kim, B. W., Kim, B. S., & Joo, S.-S. (2009). An efficient remote code update mechanism for wireless sensor networks. In *Proceedings of the IEEE Military Communications Conference* (pp. 1-7).

Mtt, L., Suhonen, J., Laukkarinen, T., Hmlinen, T., & Hnnikinen, M. (2010). Program image dissemination protocol for low-energy multihop wireless sensor networks. In *Proceedings of the International Symposium on System on Chip* (pp. 133-138).

Nordic Semiconductors. (2007). *nRF24L01 product specification*. Retrieved from http://www.nordicsemi.com/

Reijers, N., & Langendoen, K. (2003). Efficient code distribution in wireless sensor networks. In *Proceedings of the 2nd ACM International Conference on Wireless Sensor Networks and Applications* (pp. 60-67).

Stathopoulos, T., Heidemann, J., Estrin, D., & SENSING, C. U. (2003). *A remote code update mechanism for wireless sensor networks*. Retrieved from http://www.isi.edu/~johnh/PAPERS/Stathopoulos03b.html

Suhonen, J., Kuorilehto, M., Hännikäinen, M., & Hämäläinen, T. (2006). Cost-aware dynamic routing protocol for wireless sensor networks - design and prototype experiments. In *Proceedings of the IEEE 17th International Symposium on Personal, Indoor and Mobile Radio Communications* (pp. 1-5).

Wang, Q., Zhu, Y., & Cheng, L. (2006). Reprogramming wireless sensor networks: Challenges and approaches. *IEEE Network*, *20*(3), 48–55. doi:10.1109/MNET.2006.1637932

Zhang, C., Yu, Q., Huang, X., & Yang, C. (2008). An RC4-based lightweight security protocol for resource-constrained communications. In *Proceedings of the 11th IEEE International Conference on Computational Science and Engineering Workshops* (pp. 133-140).

Teemu Laukkarinen received the M.Sc. degree in computer science from Tampere University of Technology (TUT) in 2010. He is currently pursuing towards PhD in the Department of Computer Systems at TUT. His research interests include operating systems, applications and high level abstractions in wireless sensor networks.

Lasse Määttä received the MSc degree in computer science from Tampere University of Technology (TUT) in 2010. He is currently working as a researcher in the DACI research group in the Department of Computer Systems at TUT. His research interests include software design for low-power wireless sensor networks.

Jukka Suhonen received the MSc degree in computer science from the Tampere University of Technology (TUT), Finland in 2004. He is currently pursuing his Ph.D. in the Department of Computer Systems at TUT. His research interests include wireless networking, network protocol and algorithm design, and Quality of Service issues in wireless networks.

Timo D. Hämäläinen received the MSc degree in electrical engineering from Tampere University of Technology (TUT), Finland, in 1993, and PhD degree in electrical engineering from TUT, Finland, in 1997. He is a Professor at Department of Computer Systems (DCS) at TUT since 2001. He is author of over 60 journal and 200 conference publications and holds several patents. His research interests include wireless sensor networks, parallel system-on-chip architectures and design tools.

Marko Hännikäinen received his MSc in information technology in 1998, and PhD in information technology in 2002 from the Tampere University of Technology (TUT). He was nominated an adjunct professor in 2005, and since 2007 he has been a Professor at the Department of Computer Systems, TUT. He has authored over 120 publications and holds several patents. His research interests include wireless sensor networks, new wireless applications concepts, and model-based system design.

PUBLICATION 3

T. Laukkarinen, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "Pilot studies of wireless sensor networks: Practical experiences," *Conference on Design and Architectures for Signal and Image Processing (DASIP)*, pp. 1–8, November 2–4, 2011, Tampere, Finland. doi:10.1109/DASIP.2011.6136867

PILOT STUDIES OF WIRELESS SENSOR NETWORKS: PRACTICAL EXPERIENCES

Teemu Laukkarinen, Jukka Suhonen, Timo D. Hämäläinen, Marko Hännikäinen

{teemu.laukkarinen, jukka.suhonen, timo.d.hamalainen, marko.hannikainen}@tut.fi
Tampere University of Technology, Department of Computer Systems
P.O.Box 553, FI-33101 Tampere, Finland

ABSTRACT

For enabling successful field pilots of Wireless Sensor Network (WSN) applications, the network reliability and prototype testing become limiting factors. Application pilot studies need to operate end-to-end, covering the physical durability of devices, embedded software, and infrastructure interfaces and data collection. This paper summarizes our pilot study experiences, and what tools and practices were required. Six lessons are proposed: a systematic pilot template results straightforward pilot completion; shared WSN infrastructure reduces labor; tailored embedded software testing tools are needed; the pilot must be prepared carefully; the WSN technology must be usable for research partners; and the pilot must be maintained and maintenance tools are required in large scale pilots. Our experiences base on over 20 pilot studies and over 1000 deployed devices. This paper describes 11 main pilots, which utilize from 10 to 377 devices per pilot.

Index Terms— wireless sensor networks, sensor systems and applications, pilot study

1. INTRODUCTION

A Wireless Sensor Network (WSN) consists of several measuring devices (nodes), which autonomously form a communication network [1]. The nodes sense their environment, process and store data, and deliver data towards the sinks. These sinks act as gateways to other networks or infrastructure servers.

WSNs are widely studied, as they are an enabling technology for numerous ubiquitous applications in different fields, such as environmental and energy monitoring, building automation, and security. WSN technology is composed of complex distributed embedded systems. Typically, the targeted WSN nodes are small and low-cost, and operate years with small batteries or use solar panels. Thus, the technology is resource constrained in computational power, communications capacity, memory, and in energy.

WSN development has been relying for field piloting to achieve reliable performance and application feasibility results [2]. In practice, prototype applications are implemented on a selected WSN technology. For enabling successful field pilots of WSN applications, the networking reliability and prototype testing become limiting factors. Application pilots operate end-to-end, covering the physical durability of devices, embedded software, and infrastructure interfaces and data collection.

The resource constraints and demanding nature of pilot installations make the prototype testing and the management of pilot reliability difficult. Operability needs to be achieved on node level functionality, network level operations, and application level [2]. A laboratory testing with a small number of nodes is not enough to ensure application functioning in real conditions [2], as constantly changing environmental conditions affect radio communication and hardware durability.

TUTWSN [3] is a WSN prototype technology developed at the Department of Computer Systems, Tampere University of Technology. TUTWSN has been used in over 20 field pilots with over 1000 nodes between 2007 and present. Field pilots have been organized with research partners and they have been concentrating on certain applications, such as building automation measurements, transport logistics, and personnel security.

TUTWSN prototypes are battery powered, embedded sensing devices operating on 2.4 GHz and 433 MHz ISM radio bands. The technology consists of hardware platforms, multihop mesh protocol stack, and end-to-end application infrastructure. TUTWSN represents a modern resource constrained battery powered WSN technology.

As a summary of pilot experiences, we present six lessons, which we consider the most important lessons in relatively large WSN pilot studies. A systematic pilot template is proposed for straightforward pilot study completion. A shared WSN infrastructure is shown, which can be cloned for parallel pilots to reduce labor and speed up piloting. Tools for difficult embedded software testing are described. Purpose of the tools is to ensure WSN functionality before the pilot. Required preparations before a pilot are described, such as a testing checklist. The role of partners must be taken into account. This requires deployment tools and integration support. Finally, the pilot must be maintained, which requires tools in large pilots.

The experiences are compared with four main related lessons publications. Langendoen et al. [4] present a troubled pilot study of potato field monitoring with 109 nodes. This is one the first lessons learned type of publications of WSN pilot studies. Barrenetxea et al. [2] present seven environment pilot studies with 179 nodes. Corke et al. present nine iterative environmental monitoring pilot studies with 355 nodes in [5] and provide lessons, which led to an end-to-end WSN solution for environmental monitoring. A water quality monitoring network in a sugar cane farm is presented by Hu et al. in [6] with long range nodes (up to 1 km). These four publications give several lessons of pilot studies for the scientific community.

The contribution of this paper is to present experiences of pilot studies: practices and tools are proposed for pilot preparation, testing, deployment, monitoring, and maintenance. Compared to the related work, the experiences come from relatively larger pilots. These extend and confirm the work in related papers.

This paper is constructed as follows. First, the related work is settled by examining the pilot study lessons of others in Section 2. Then, the TUTWSN infrastructure is presented in Section 3. Section 4 presents the major pilots deployed with TUTWSN. The lessons from those pilot study experiences are given in Section 5. Finally, the paper is concluded in Section 6.

Table 1: Experiences and lessons described by the related research. X denotes that publication discusses the lesson on varying detail.

Experience	**Langendoen et al. [4]**	**Barrenetxea et al. [2]**	**Corke et al. [5]**	**Hu et al. [6]**
Difficult embedded software testing	X	X	X	X
Problems with backbone network connections (GPRS/3G etc.)	-	X	-	X
Problems with node enclosures	X	X	X	-
Challenging software complexity	X	X	X	X
Watchdog required and/or useful	-	X	X	X
Deployment monitoring, controlling and/or tracing	X	X	X	-
Installation and/or deployment tools	-	X	X	-
Importance of partners	-	X	X	-

2. RELATED WORK

The related work concentrates on pilot study experiences of the mentioned four main related publications [2, 4, 5, 6]. Table 1 gathers reported experiences, which are mentioned at least in two of these publications.

Difficult embedded software testing is constantly reported. Malfunctioning nodes cause trouble in deployment and if there is no tracing in deployed nodes, the problem is difficult to solve.

Typically, each deployment uses some server infrastructure for data storage. In remote deployments, an Internet connection is required. 3G/GPRS etc. modems are used, however, there are often problems with these. Node enclosures are also a difficult task. If the nodes need to be accessed during the deployment, the enclosure cannot be fully hermetic.

WSNs require complex software, e.g. protocol stacks. Complexity makes development challenging. Watchdogs have been useful on various levels, from nodes to servers. The WSN should be remotely monitored and controlled during the deployment. Tracing is required for the later analysis of possible problems. Deployments require tools for installation and maintenance.

Finally, partnering with other researches is vital, since one cannot master everything. Each of these lessons presented in Table 1 is covered in more detail in the lessons section.

Development experiences of ZigBee from standard to commercial products are presented in [7]. The main contribution compared to other publications is the necessity of simple installation tools. Two tools are described, which are courtesy of Eaton's Home Heartbeat product. First, a simple name insertion for a sensor node is described. When a node is activated with a key of a key chain fob, the name can be selected from the LCD screen in the key fob. Second, instead of full mesh network, Home Heartbeat uses range-extenders and the actual sensor devices are non-routing. This is easy to understand and install for everyone. These are similar approaches to our memorable IDs and deployment tool. However, we managed to make an easy-to-use deployment tool for a full mesh network. The tool is presented in Section 5.5.

WSN deployments have been reported constantly on various applications, such as habitat monitoring [8], volcano monitoring [9], semiconductor plant and oil tank monitoring [10], and microclimate monitoring of a redwood [11]. These papers present the application, measurements, and experiences of the WSN technology. Emphasis is either on the application and measurements or on the WSN technology functionality. These papers provide technical lessons about protocols and hardware, whereas this paper targets to end-to-end studies.

3. TUTWSN INFRASTRUCTURE

TUTWSN is a full mesh network with multihop and ad-hoc features. It is autonomous and does not require configuration when deployed. The TUTWSN protocol stack consists of four layers: physical, low energy consumption [3, 12] and low latency [13] Medium Access Control (MAC) and routing layers, and application layer [14]. Also, systems software have been developed: operating system [15, 16], middleware [17] and independent firmware update protocol with a bootloader [18].

The TUTWSN hardware consists of three main boards, which share the same micro controller unit, and temperature and luminance sensors. Accelerometer and humidity sensors are optional. There is a connector for external sensors. This connector has pins for digital I/O, analog-to-digital converter input, I2C bus, SPI bus and UART interface. 2.4 GHz and 433 Mhz off-the-shelf radio transceivers are used. AA or AAA batteries are typically used as an energy source. A mains transformer or a solar panel are optional sources.

Ethernet gateways have additional UART-to-Ethernet bridge and an SD memory card slot for buffering data during possible Ethernet connection outtakes.

A server infrastructure is used with TUTWSN. It consists of a gateway software with a database. A Java user interface software, mobile web user interface, and SMS messages are provided for end users. Several server interfaces are provided for integration to other systems. As a result, TUTWSN is an end-to-end WSN infrastructure, which is presented in Figure 1.

4. MAIN TUTWSN PILOTS

Pilots have tested TUTWSN on various environmental conditions from subzero temperatures on outdoors to high humidity and temperatures in sauna. Furthermore, pilots have tested quick installation, small and large networks, and various sensors for different usage. Finally, pilots provided knowledge of WSN usage in applications and usable data for the end user.

We have piloted TUTWSN with several research and industry partners. The main pilots are gathered in Table 2 and are explained in the following sections. The pilots are briefly introduced and given as a background for the lessons in the Section 5.

4.1. Sewer water level monitoring

In the sewer line monitoring, the problem was a waste water line of a chemical factory, which runs through the city and occasionally flooded to the streets. This water line was around 5 km long.

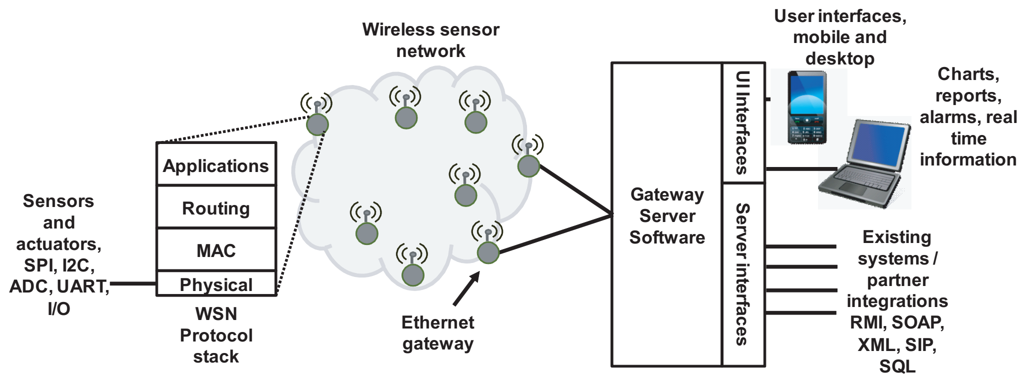

Wireless sensor network
User interfaces, mobile and desktop
Charts, reports, alarms, real time information
Applications
Routing
MAC
Physical
WSN Protocol stack
Sensors and actuators, SPI, I2C, ADC, UART, I/O
Ethernet gateway
Gateway Server Software
UI Interfaces
Server interfaces
Existing systems / partner integrations RMI, SOAP, XML, SIP, SQL

Fig. 1: TUTWSN infrastructure. It starts from the node sensors and actuators, and ends to the end user interfaces.

Table 2: The main pilot studies deployed with TUTWSN. The hospital personnel safety pilot used low latency MAC and routing while the other pilots used low energy consumption MAC and routing.

Pilot Study	Nodes	Duration	Technology
Sewer water level monitoring	25	2009-	433 MHz
Chemical factory monitoring	62	2009 (a year)	2.4 GHz
Green house temperature and humidity leveling	30	2009 (a month)	2.4 GHz
Campus network for teaching	340	2008-	2.4 GHz
Home monitoring in several homes	180	2007-	2.4 GHz
Transportation cargo monitoring	10	2009 (a year)	2.4 GHz
Building monitoring	377	2008-2010	2.4 GHz
Environment monitoring	60	2005-	433 MHz
Environment monitoring, ground frost and snow depth	30	2007 (a year)	433 MHz
Cattle living conditions in barn	30	2009-	2.4 GHz
Hospital personnel safety	62	2009-	2.4 GHz

The goals of the sewer water level monitoring pilot were: 1) Equip wells in a 5 kilometers long water line with water level sensors to monitor water level. 2) Test WSN functionality and performance in long-range alarming application. 3) Study the benefits of WSN installation speed and cost-efficiency.

In the installation, every well was equipped with a TUTWSN node, two water level sensors, and a temperature sensor. A line of extra routing TUTWSN nodes was constructed above the ground. Both ends of the line were equipped with gateways to the Internet to improve robustness. Data was gathered to a server and integrated into the existing factory system. The installation had 23 battery operated long range nodes (433 MHz) and two mains powered gateways. One maintenance visit was needed during the deployment due to the failed sewer nodes.

The experiences of this pilot study were: 1) The first sewer nodes did not withstand the very high humidity and rapid temperature variations of the wells. They had to be replaced with hermetically closed enclosures. This is presented in Figure 2. 2) The deployment tool was helpful for constructing reliable routing line. 3) A new server interface was needed for integrating TUTWSN into the existing systems. 5) A WSN functionality monitor was needed to alarm from possible failures.

4.2. Chemical factory monitoring

Chemical factory monitoring focused on two main cases: monitoring fault connections in the factory and monitoring loading of trucks at the premises. The goals were: 1) To study WSN functioning inside a factory, which is full of metal pipes and other metal structures. 2) To receive alarms from fault connections. 3) To measure loading levers positions and track trucks inside the factory premises.

The installation consisted of 60 nodes (2.4 GHz) and 2 gateways. The network was installed inside the factory and to the outside premises. In addition, few mobile nodes were placed in the trucks.

The experience of this pilot study was that a new server interface was needed for integrating TUTWSN into the existing systems.

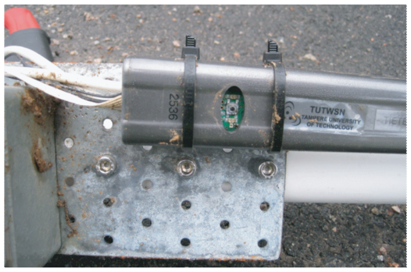
4.3. Greenhouse temperature and humidity leveling

In Greenhouse monitoring, the network was installed temporarily in two greenhouses and a storage space. The goals were: 1) To find out temperature and humidity leveling the greenhouse spaces. 2) To study the installation speed of the WSN. 3) To study the benefits of the WSN for growing plants.

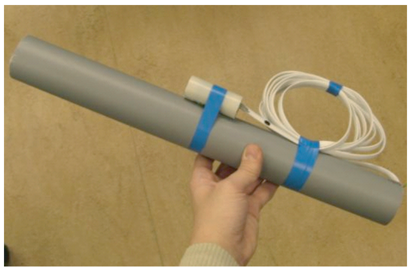
The installation consisting of 28 nodes (2.4 GHz) and 2 gateways took one day. The node constellation is shown in Figure 3. During the deployment, the staff of the greenhouse studied temperature and humidity graphs and noticed that a part of the second greenhouse was significantly colder. They adjusted the air conditioning and added insulation according to the measurements. As a result, the temperature leveled across the greenhouse.

The experiences of this pilot study show that the clonable WSN infrastructure and automated configuration reduced the workload and allowed a quick and short pilot.

(a) A node in a tree at -20 Celsius degree frost.



(b) A destroyed sewer node. Circuit board and battery holders were corroded due to the high humidity and condensation.



(c) A robust sewer node, which is molded in epoxy resin inside a PVC pipe.

Fig. 2: Node pictures in various conditions. Pilot will not succeed, if the enclosing is not prepared according to local conditions.

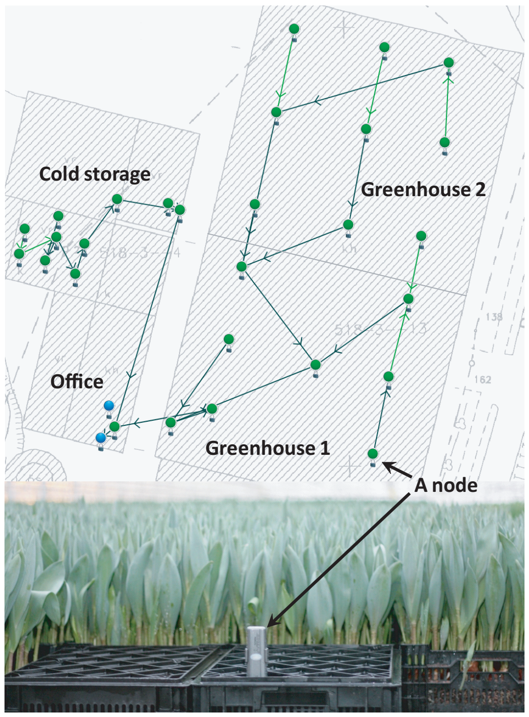


Fig. 3: The installation constellation and an installation picture of the greenhouse monitoring pilot. A circle represents a node and an arrow between circles represents the route of the data.

4.4. Campus network for teaching

Our largest single deployment so far has been the campus network at the Tampere University of Technology campus. The goals were: 1) To study TUTWSN functionality in a large deployment. 2) To create a WSN platform for students to study and to create applications for. 3) To provide meaningful measurements for the facility management.

At the moment, the installation consists of 340 nodes, which are installed in the public areas of the campus. The network is used on a course, where students can develop their own applications on top of the network. Furthermore, the network is used as research test-bed [18].

The experiences of this pilot study were: 1) 340 nodes require automated configuration and updating. 2) Node logistics must be precise in a such large scale.

4.5. Home monitoring in several homes

For home monitoring the goals were: 1) See the benefits of WSNs for home users. 2) Study the installation easiness of the WSN.

The installation was done by the home owners. Networks of 10-40 nodes were delivered to over 10 homes. These networks measured temperature, humidity, illumination and CO2 in various rooms, including sauna. Furthermore, passive infrared sensors, magnet switches, and piezo-electromechanical pressure sensors reported of unauthorized access.

The experiences of this pilot study were: 1) Labor amount was not significant due to WSN infrastructure cloning although there were over 10 parallel pilot locations. 2) Installing a mesh network requires a deployment tool. 3) People want simple physical identifiers for the nodes.

4.6. Transportation cargo monitoring

The goals were to test WSN functionality in a truck, gather useful information from the cargo, track the truck, and identify attached trailers.

The installation was done by the partner. A cargo space of a truck and its trailers were monitored with 10 nodes. Temperature, humidity and acceleration were measured. A notification of every over four g-force impact was sent. Furthermore, a GPS tracking of the truck was used. The attached trailer was identified from the node neighbor information.

The experience of this pilot study was that the battery holder springs were too soft to withstand over 4 g-force impacts, and the node briefly lost its supply current. This was noticed in the preliminary checks. The springs were replaced with solid contacts.

4.7. Building monitoring

Building monitoring pilots have concentrated on the working conditions of employees. The goals were: 1) To measure indoor air quality with WSN from multiple points. 2) To monitor room usage with the WSN. 3) To study WSN benefits for installing to existing buildings.

The installations were performed by our partners. Temperature, humidity and CO2 levels were measured. Also, room usage was monitored with CO2 and passive infrared sensors.

The experience of this pilot study was that installing a mesh network requires a deployment tool.

4.8. Environmental monitoring

An outdoor temperature and humidity environmental monitoring network was deployed to test long-range 433 MHz TUTWSN. The goals were: 1) Test TUTWSN functionality in outdoor environment with long-range hops. 2) Provide measurement data for local farmers and other interest groups.

The installation covers approx. six square kilometers with 60 nodes. It measures luminance, air temperature and humidity, ground temperature and moisture, and lake temperature. A web user-interface was developed for the measurement data. The pilot has been running since 2005 and it has been updated several times.

4.9. Environmental monitoring, ground frost and snow depth

The goals were: 1) The partner wanted to test WSN technology. 2) Ground frost and snow depth were measured with temperature sensors.

The installation took place in Lapland with 30 nodes. Ground frost and snow depth measurements based on multi-point temperature measurements, where depths were determined from the temperature. Sensors were attached to one cable with 10 cm between each sensor. For the ground frost measurement, a hole was dug and the sensors were set in to that hole. Sensors were hanged on air for the snow depth.

The experience of this pilot study was that lacquering the node circuit board for protection is enough for outdoor nodes.

4.10. Cattle living conditions in a barn

The goals were: 1) Test WSN technology in a barn. 2) Measure the living conditions of the cattle. 3) Monitor the security of the barn.

The installation in the barn comprised 30 nodes. Temperature, humidity, CO2 and luminance were the main measurements. In addition, passive infrared motion detectors and magnet switches were deployed to monitor the security of the barn and safety of the cattle. A water pressure switch was integrated into TUTWSN. It monitors the pressure of the cattle drinking water supply system. If the pressure drops, an alarm is sent.

4.11. Hospital personnel safety

The goals were: 1) Locate a person triggering the alarm from hospital at room level accuracy. 2) Test the low latency WSN.

Hospital personnel safety pilot study is presented in detail in [13]. The installation consists of nine sink nodes, 41 routing nodes and 12 mobile nodes. Personnel carry mobile nodes, which are continuously localized. An alarm can be activated from the mobile node when the person encounters a threatening situation. Then, the user interface shows the room, where the alarm sender is. In addition, the network delivers measurements and actuator commands.

The experience concerning this paper is the cloned WSN infrastructure. Different MAC and routing layers were used compared to other presented pilots. However, the WSN infrastructure was otherwise the same.

5. LESSONS

This section presents our experiences compared to the four related pilot study articles [2, 4, 5, 6].

5.1. Lesson 1. A systematic pilot plan is needed

Each of our pilots followed the same template. First, goals for the pilot were determined. This was done in close collaboration with our research partners. In addition to the goals, the required measurements and actuators, the deployment size, the expected deployment lifetime, and the schedule were settled. Second, the pilot was executed. This includes development, installation, and maintenance. Finally, the results of the pilot were drawn according to the original goals. This is a straightforward template to complete pilot studies.

Common goals for the pilots were testing the WSN suitability, possible technology benefits, and functioning in the application. Typically, each pilot required at least one new measurement. This required integration of new sensors to TUTWSN before deployment. A pilot plan document was created, which defined the goals, required new measurements, deployment size, and expected results of the pilot.

The pilot execution was divided into three phases. First, new sensors were integrated, nodes were assembled, and preliminary tests were performed. Then, the deployment was conducted with research partners. Finally, the pilot was monitored and maintained.

The pilot results were gathered after the agreed deployment time. The original goals were compared to the deployment execution. The collected results were disseminated to the pilot partners.

5.2. Lesson 2. WSN infrastructure must be shared by parallel pilots

Corke et al. presented an end-to-end WSN architecture as a result of several pilots in [5]. They had five different hardware, software, and network protocol combinations for their pilots, which eventually led to the end-to-end solution. All our pilots shared the same TUTWSN infrastructure. This was vital to reduce labor and management of the pilots and removed the need for extensive preliminary testing before each pilot.

Since we had a tested technology, the development before a pilot required only new sensor integrations. Adding a new sensor required a driver and a packet description development on to the node. Packet description had to be mapped in to the database on the server. After that the sensor was immediately usable in the architecture. This repeated often in our pilots, so we created a process definition for adding a new sensor. The process defined phases required for the integration. Also, it defined required new source code files for the new sensor.

As [5] states, end-to-end solution automates the recurring tasks of the WSN deployments. In order to conduct many parallel pilots, we needed the shared WSN infrastructure to reduce labor and speed up the start of the pilots.

5.3. Lesson 3. Tailored testing tools are required for the embedded WSN software

All four reference publications note the importance of software testing for successful deployment [4, 2, 5, 6]. WSNs are considered difficult to test, since they are resource constrained embedded devices and distributed [19]. There are no verbose debugging interfaces and the distribution requires debugging of several devices at once.

We had to develop tools for testing the embedded code, before the sufficient level of functionality was achieved for the pilot studies. These tools were run-time assertions with stack trace, static analyzers, a distributed debug printing interface, and a network sniffer.

The run-time assertions check that embedded code is working properly and prerequisites are not violated. If a failing assertion is encountered, the node will send file and line information about the error over radio, write it on to EEPROM, and print it through UART. The information can be caught with a sniffer, read with an EEPROM reading device, or received to a terminal through UART. Also, the stack trace is delivered with the same methods. This information help tracing the origins of the bug. The assertions resemble ones presented in [20].

Static analyzers find the hazardous points of the embedded C code. We have used an open source Splint analyzer to ensure higher code quality. A static analyzer for the stack size consumption was needed, since the small MCU on a node has a limited function depth. If the stack overflows, the code might tangle. Due to the complex protocol stack, this has happened to us occasionally before using the static code analyzer. Corke et al. reported a stack analyzer tool for optimizing thread stack sizes to save memory [5].

The distributed debug printing interface allows us to enable and disable debug prints in the embedded code. TUTWSN nodes and sinks are basically the same devices. Sinks are extended with an UART-to-Ethernet adapter. We used these sinks as nodes in testing. This allowed a distributed collection of debug prints to a server, which solved efficiently the problem of testing small embedded distributed devices.

Finally, the network sniffer gives an instant look to the protocol stack behavior on the field. It presents captured traffic in a human readable form and illustrates time and frequency division scheduling of the MAC layer. The sniffer allowed us to trace long term misbehaviors and find out if the MAC layer was operating according to the designed algorithms. Authors in [4] stated the necessity for such packet sniffer and in [2] presented a sniffer tool that presents measurement values on the field.

This testing tool set allowed us to find all major flaws from the embedded WSN code. Figure 4 presents an availability graph [21] of our protocol stack improvements in one test iteration. The availability graph shows the probability to receive a sample within the time interval. The dashed graphs show the worst and the average reception interval before updating the tested and improved protocol stack. The solid lines represent the intervals after updating. The updated worst node achieves the average of the older.

As [4] and [2] state, the protocol stack testing and functional verifying are a key factor for successful pilot study. In [2], authors state that things should be kept as simple as possible to achieve this goal. However, the presented tools can significantly help testing even complex embedded WSN software.

5.4. Lesson 4. The preparation is important before the pilot

We had a checklist, which was reviewed before deploying the network. This checklist mainly consisted of basic functionalities (e.g. every node delivers correct data from every sensor), where the network was quickly put up on a desk and checked. In addition, the energy consumption of the nodes was continuously monitored. Adding a new sensor to the node is safe for the network operations, but it is easy to leave some I/O pin of the micro controller unit on a wrong position. One such pin can increase energy consumption significantly.

WSNs are often used in harsh conditions and protecting the embedded hardware has been difficult as stated by three reference papers [2, 4, 5]. Rain, condense water, and extreme temperatures are typically encountered conditions. We used lacquer to protect circuit boards from moisture in simple outdoor installations. In fact, this was enough for all our outdoor pilots, except for the sewer nodes. Figure 2b shows a destroyed sewer node. The sewer had a very high humidity and rapid temperature changes. If the conditions are very

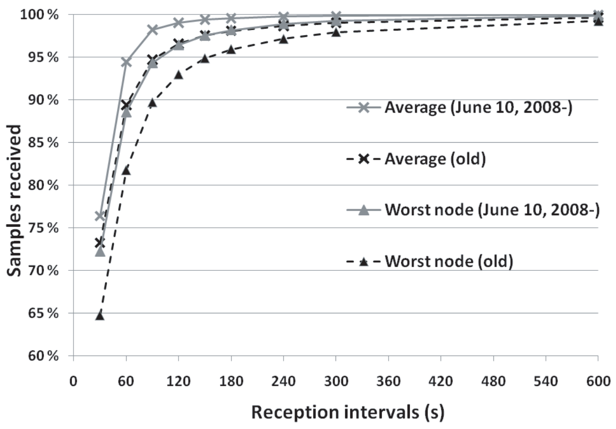


Fig. 4: An availability graph presents improvements achieved in one iteration of testing with presented tools. 95% availability is achieved by average in 60 seconds after update. Before update, 95% availability was achieved in 90 seconds.

harsh, enclosing a node cannot be overdone. We molded nodes in to resin inside a PVC pipe. Figure 2c shows such sewer node. Two D batteries were used to ensure sufficient life-time, since such node is usable only once. D batteries have approx. ten times the capacity of AA batteries.

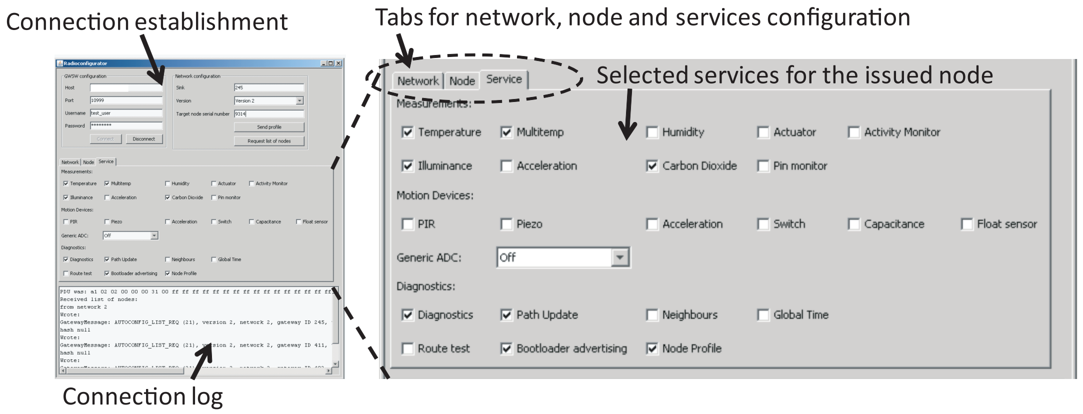
Enclosing is an often reported problem. Molding nodes hermetically protects them from moisture and corrosion. However, as the hardware is no longer accessible, the WSN must be thoroughly tested. Our pipe sewer nodes have been functioning for over one and half years without maintenance.

Backbone server connection problems are reported in [2] and [6]. We have encountered the same problem in two forms. First, the partner had a high network security and despite the preparations, correct ports had not been opened at the time of deployment. Second, the wireless Internet connection (GPRS/3G, Flash-OFDM or satellite) had a poor coverage at the deployment site. In our experiences, taking different connection options to the deployment site is important. This ensures that the deployment is got running, even if the partner network is not yet ready or the service of one wireless Internet connection is poor. When the deployment site is in a remote location, this saves time and money.

5.5. Lesson 5. The technology must be usable for the research partners

The gathered data is useless as itself, someone must know what to do with the data [2]. Thus, it is wise to partner with experts of the application area. Most of our pilot studies were originally composed by industry or research partners. We concentrated on providing the technology for the partners and the partners were responsible for interpreting data correctly. Furthermore, the partners typically wanted to do application case studies or exploration of WSN possibilities in their application area. The goal was to see benefits and usability of WSN in the studied application. This often required integration to the existing systems.

Our architecture provides several interfaces, which can be used to integrate TUTWSN as a part of other systems. However, some existing systems required new interfaces, for example XML SOAP, Java RMI and OPC connectivity. Our server software has a plug-in architecture, which allows extension with new interfaces.



GWSW configuration
Host:
Port: 10999
Username: test_user
Password: ********

Network configuration
Sink: 245
Version: Version 2
Target node serial number: 9314

Measurements:
[✓] Temperature [✓] Multitemp [ ] Humidity [ ] Actuator [ ] Activity Monitor
[✓] Illuminance [ ] Acceleration [✓] Carbon Dioxide [ ] Pin monitor

Motion Devices:
[ ] PIR [ ] Piezo [ ] Acceleration [ ] Switch [ ] Capacitance [ ] Float sensor

Generic ADC: Off

Diagnostics:
[✓] Diagnostics [✓] Path Update [ ] Neighbours [ ] Global Time
[ ] Route test [✓] Bootloader advertising [✓] Node Profile

```
PDU was: a1 02 02 00 00 00 31 00 ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff
Received list of nodes:
from network 2
Wrote:
GatewayMessage: AUTOCONFIG_LIST_REQ (21), version 2, network 2, gateway ID 245,
hash null
Wrote:
GatewayMessage: AUTOCONFIG_LIST_REQ (21), version 2, network 2, gateway ID 411,
hash null
Wrote:
```

Fig. 5: A screenshot of auto-configurator. It can configure network, node, and services configurations to a node wirelessly.

Occasionally, the deployment was installed by a partner. Installing an autonomous mesh WSN should be easy. However, limited node amount and difficult to understand operation can make installation difficult even to an experienced engineer. We created a deployment tool that can be activated with a push of the press button. The node constantly scans the network and indicates immediately with LEDs if the current location is a good installation point. The deployment tool is deactivated with another push of the button or by itself after a certain time period. Energy is not wasted, since LEDs are used only for a short period of time, as LED energy consumption was discussed in [2]. We instructed to start the installation from the sink and advance there on. As a result, a network can be installed rapidly by anyone, and it is guaranteed to function well as long as the installer follows the LED user interface.

5.6. Lesson 6. The pilot must be maintained and maintenance tools are a necessity in large scale pilots.

WSNs are considered data-centric, it does not matter which device delivers the required data from the location. However, as [2] states, tracing the data to the original source is often required. For example, if a sensor fails, it must be replaced in most cases. In [2] every measurement packet were labeled with an ID. In our network, each packet contains a node ID. Thus, tracing the source is trivial. The problem is that when over 1000 nodes are deployed, it is a demanding task to keep up with network and node configurations. We had to develop two tools for configuration and tracing of the nodes in the pilots.

An auto-configurator was developed for an easy node configuration. Nodes were programmed with a bootloader and a program image at the factory. When the node is deployed (e.g. it is assigned to a network and required sensors are attached to it), the required configurations are set with the auto-configurator *wirelessly*. This allows a fast deployment configuration. The auto-configurator user interface is presented in Figure 5. It has a connection establishment part, configurations for a network, a node, and services, and a status logging window.

The auto-configurator enables effective network wide programming, which is close to the requirement for network programming presented by Corke et al. at the discussion part [5]. The approach is still per-node programming, but the same program image is used on all nodes in the network. This simplifies the development of the large scale deployments, what Corke et al. demand.

Each node has a globally unique serial number which is set at the factory. In the auto-configuration, a logical node ID is given to each node. These are unique in one network. This scheme allows us to use simple IDs. In our experience, users want to see node IDs and they want them to be short. Thus, user can see the same number in the user interface and in the physical node on a pilot although this is not the data-centric idea. User can set a name for each node in the user interface and hide the ID. The IDs are used only for installation and maintenance to distinguish the physical node. Physical traceability is noted as a necessity in [2].

Nodes need to be replaced occasionally. In addition to the auto-configurator, a web service was developed, which stores serial number, configuration, and software version of each node. If a node fails and requires replacing, a new factory programmed node can be configured to replace the malfunctioning one. The new node may have the same logical node ID. This same web service tracks deployed networks and provides directly server software configurations. This tool was a necessity, when the node and pilot amounts increased. Corke et al. discuss these practical issues as a not fully explored research area [5].

In addition to node maintenance, automatic controlling and monitoring are useful tools for maintenance [2]. Our controlling is limited to automated data requests from the network. Otherwise the network is autonomous. For monitoring, we have automated server monitors for Internet connection between gateway nodes and server software, for malfunctioning nodes, and for server software and hardware. If something fails, an email or a SMS message is sent to persons with interest. These are a necessity on pilots, which monitor critical systems, such as the sewer water level monitoring network. Detecting node malfunctions and recovering from these faults is considered major challenge in [5] for large scale deployments.

As the authors in [2] state, the WSN deployments are not a case of leave and forget. Earlier experience are backed up by our experience that despite the data-centric approach, WSNs require traceability. With the presented approach, it is easy to trace faulty measurements and failures to nodes and sensors. Further, replacing a node does not require dramatic actions or skills of an engineer. Finally, the end-user has a continuous set of memorable IDs. This helps users, which are not familiar with the data-centric operation.

6. CONCLUSIONS

This paper presented six practical lessons of WSN pilot studies. A systematic WSN pilot study template was proposed for the straightforward completion of the pilots. Benefits of a shared WSN infrastructure were described for parallel pilots. Required testing tools were presented for embedded WSN software testing. The importance of pilot preparations was noted. Required actions for easy partnering were presented. Finally, the maintenance of large scale pilots was discussed and maintenance tools were presented. Each lesson presented our experiences and compared them to four reference publications. A successful pilot must function reliably from end-to-end, must integrate into the systems of partners, and must be maintained, even if the node amounts are relatively large.

7. REFERENCES

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

[2] G. Barrenetxea, F. Ingelrest, G. Schaefer, and M. Vetterli, "The hitchhiker's guide to successful wireless sensor network deployments," in *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, New York, NY, USA, 2008, pp. 43–56, ACM.

[3] M. Kohvakka, J. Suhonen, T. Hämäläinen, and M Hännikäinen, "Energy-Efficient Reservation-Based Medium Access Control Protocol for Wireless Sensor Networks," *EURASIP Journal on Wireless Communications and Networking*, vol. 2010, pp. 22, 2010.

[4] K. Langendoen, A. Baggio, and O. Visser, "Murphy loves potatoes: experiences from a pilot sensor network deployment in precision agriculture," in *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, apr. 2006, p. 8 pp.

[5] P. Corke, T. Wark, R. Jurdak, Wen Hu, P. Valencia, and D. Moore, "Environmental wireless sensor networks," *Proceedings of the IEEE*, vol. 98, no. 11, pp. 1903 –1917, nov. 2010.

[6] W. Hu, T. Dinh, P. Corke, and s. Jha, "Design and deployment of long-term outdoor sensornets: Experiences from a sugar farm," *Pervasive Computing, IEEE*, vol. PP, no. 99, pp. 1 –1, 2010.

[7] A. Wheeler, "Commercial applications of wireless sensor networks using ZigBee," *Communications Magazine, IEEE*, vol. 45, no. 4, pp. 70–77, 2007.

[8] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, New York, NY, USA, 2002, WSNA '02, pp. 88–97, ACM.

[9] G. Werner-Allen, K. Lorincz, M. Ruiz, O. Marcillo, J. Johnson, J. Lees, and M. Welsh, "Deploying a wireless sensor network on an active volcano," *Internet Computing, IEEE*, vol. 10, no. 2, pp. 18 – 25, 2006.

[10] L. Krishnamurthy, R. Adler, P. Buonadonna, J. Chhabra, M. Flanigan, N. Kushalnagar, L. Nachman, and M. Yarvis, "Design and deployment of industrial sensor networks: experiences from a semiconductor plant and the north sea," in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2005, SenSys '05, pp. 64–75, ACM.

[11] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, St. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong, "A macroscope in the redwoods," in *Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2005, SenSys '05, pp. 51–63, ACM.

[12] J. Suhonen, M. Kuorilehto, M. Hannikainen, and T.D. Hamalainen, "Cost-aware dynamic routing protocol for wireless sensor networks - design and prototype experiments," in *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, 2006, pp. 1 –5.

[13] V. Kaseva, T. D. Hämäläinen, and M. Hännikäinen, "A wireless sensor network for hospital security: from user requirements to pilot deployment," *EURASIP J. Wirel. Commun. Netw.*, vol. 2011, pp. 17:1–17:15, January 2011.

[14] J.K. Juntunen, M. Kuorilehto, M. Kohvakka, V.A. Kaseva, M. Hannikainen, and T.D. Hamalainen, "Wsn api: Application programming interface for wireless sensor networks," in *Personal, Indoor and Mobile Radio Communications, 2006 IEEE 17th International Symposium on*, 2006, pp. 1 –5.

[15] M. Kuorilehto, T. Alho, M. Hännikäinen, and T. D. Hämäläinen, "Sensoros: A new operating system for time critical wsn applications," in *Embedded Computer Systems: Architectures, Modeling, and Simulation*, Heidelberg, Germany, Aug. 2007, pp. 431–442.

[16] T. Laukkarinen, V.A. Kaseva, J. Suhonen, T.D. Hamalainen, and M. Hannikainen, "Hybridkernel: Preemptive kernel with event-driven extension for resource constrained wireless sensor networks," in *Signal Processing Systems, 2009. SiPS 2009. IEEE Workshop on*, oct. 2009, pp. 161 –166.

[17] M. Kuorilehto, M. Hännikäinen, and T.D. Hämäläinen, "A middleware for task allocation in wireless sensor networks," in *Personal, Indoor and Mobile Radio Communications, 2005. PIMRC 2005. IEEE 16th International Symposium on*, 2005, vol. 2, pp. 821 –826 Vol. 2.

[18] L. Määttä, J. Suhonen, T. Laukkarinen, M. Hännikäinen, and T. D. Hämäläinen, "Program image dissemination protocol for low-energy multihop wireless sensor networks," in *International Symposium on System-on-Chip 2010*, Tampere, Finland, Sept. 2010, pp. 129–135.

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

[20] K. Römer and M. Ringwald, "Increasing the visibility of sensor networks with passive distributed assertions," in *Proceedings of the workshop on Real-world wireless sensor networks*, New York, NY, USA, 2008, REALWSN '08, pp. 36–40, ACM.

[21] J. Suhonen, T.D. Hämäläinen, and M. Hännikäinen, "Availability and End-to-end Reliability in Low Duty Cycle Multihop Wireless Sensor Networks," *Sensors*, vol. 9, pp. 2088–2116, 2009.

PUBLICATION 4

T. Laukkarinen, J. Suhonen, and M. Hännikäinen, “A Survey of Wireless Sensor Network Abstraction for Application Development,” *International Journal of Distributed Sensor Networks*, vol. 2012, Article ID 740268, 12 pages, 2012. doi:10.1155/2012/740268

First published by Hindawi Publishing Corporation.

Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2012, Article ID 740268, 12 pages
doi:10.1155/2012/740268

Research Article

A Survey of Wireless Sensor Network Abstraction for Application Development

Teemu Laukkarinen, Jukka Suhonen, and Marko Hännikäinen

Department of Computer Systems, Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland

Correspondence should be addressed to Teemu Laukkarinen, teemu.laukkarinen@tut.fi

Received 25 June 2012; Revised 14 September 2012; Accepted 5 November 2012

Academic Editor: Arne Bröring

Wireless sensor network (WSN) application development is not an easy task due to its resource constrained nature and vast feature rich application space. Several abstractions are harnessed to ease out the difficult WSN application development. In this paper, three levels of abstractions are classified from the existing literature: node, network, and infrastructure abstractions. Since the node and network abstractions are already a well-studied area, the infrastructure abstraction is surveyed in detail to complete knowledge. Technology interoperability, service discovery, metadata support, and processing support are found as basic requirements for infrastructure abstraction. Problematic security and quality of service topics are discussed and the open research questions of ontology, service discovery, distributed processing, and performance metrics are defined. Finally, a distributed middleware design is presented as a possible solution for the key open research question: how to utilize capabilities of the abstracted technologies.

1. Introduction

A wireless sensor network (WSN) consists of even thousands of resource constrained devices (nodes), which form a distributed autonomous network [1]. Energy, computation, communication, and memory constrained WSNs must react to real world phenomena, process and fuse data, and eventually create new knowledge. This knowledge must be presented to an end-user or analyzed to create value added end-user services.

Getting data from a physical sensor to an end-user is not a simple task in WSN application development due to the resource constraints, complex protocols, and multiple levels of technologies involved in the delivery [2–4]. Therefore, different abstraction levels are needed to make application development easier. Three levels can be classified from the existing research work: node, network, and infrastructure abstractions.

The main contributions of this paper are the classification of the three abstraction levels, and a survey of the WSN infrastructure abstractions. The authors of this paper consider node and network abstractions well surveyed and defined area of the WSN research, but find a lack of definition of the infrastructure abstraction. The survey part presents the diverse field of the infrastructure abstraction and gathers a common set of requirements. In addition, we propose open research questions and present our design approach to meet some of those questions.

The paper is constructed as follows. Section 2 presents the three abstraction levels based on the existing publications. Section 3 presents the related work for this survey. Our motivation is given in Section 4. Section 5 presents the survey of infrastructure abstractions and Section 6 collects the properties of the surveyed proposals as requirements for the infrastructure abstraction. Section 7 discusses open research questions, which are derived from the survey. Our design proposals for the open questions are given in Section 8, and the paper is finally concluded in Section 9.

2. WSN Application Abstraction Levels

Three levels of abstraction for WSN applications can be classified from existing WSN research work: node, network, and infrastructure abstractions. Figure 1 positions these levels in the WSN infrastructure and each abstraction is described in detail in the following sections.

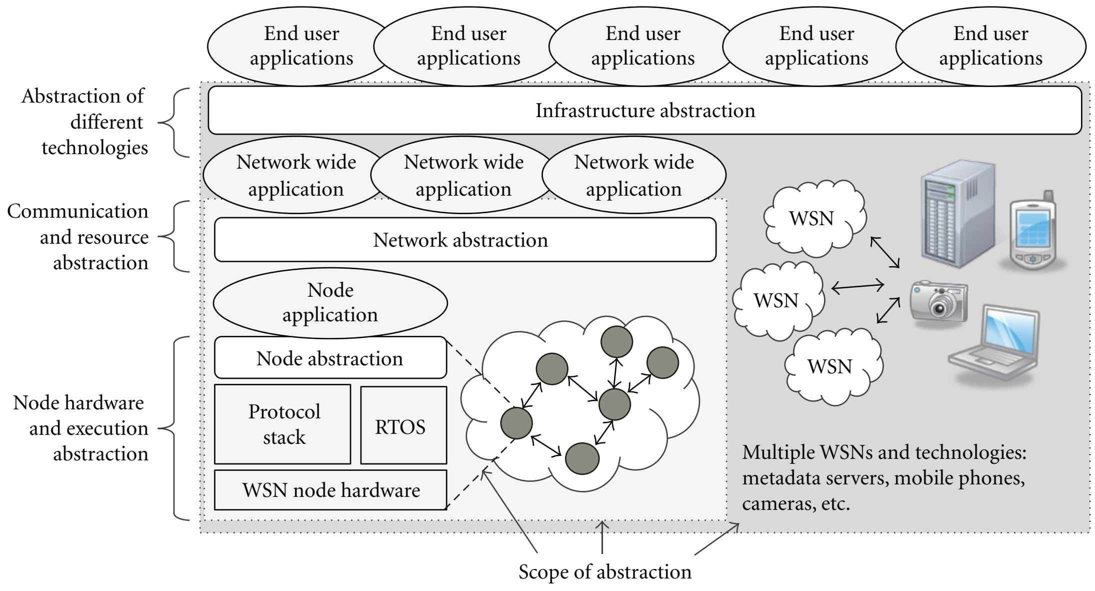


Figure 1: The abstraction levels of WSN application development.

2.1. Node Abstraction. The resource constrained embedded node hardware and communication protocols are abstracted with *node abstraction* that executes applications on each physical node [2, 3]. Embedded operating systems or virtual machines are often used approaches. For example, TinyOS [5], Contiki [6], and Maté [7] are well-known proposals of the node abstraction. The applications perform such actions as reading a sensor, processing measured data, sending data to interested parties when required, and even updating/distributing applications further in the network. The application development is typically conducted with C, NesC [8], or similar programming language, and the development is node specific.

2.2. Network Abstraction. The distributed node network is abstracted from data interested users with *network abstraction*. This level works in cooperation between distributed nodes and provides such services as data access through queries, and data processing in-network through aggregation and fusion [4, 9, 10]. Also, it can provide distribution services for node applications, such as sharing measurements through distributed memory abstraction. TinyDB [11], COUGAR [12], Agilla [13], and TinyLime [14] are often referred proposals of this abstraction level. The node and network abstractions have research history of over decade now, and several surveys have been published of them [2–4, 9, 10, 15, 16].

The network abstraction provides access to the WSN measurement data, but the measurement data itself is rarely sufficient for end-user application [4]: the measurement data can be combined with metadata (e.g., add physical location information, descriptive name, and place the measurement on a map), further processed, combined with data from other technologies, or archived for later study. Since the network abstraction and resource constrained nodes cannot provide all the required data and processing for the end-user application, an *infrastructure abstraction* is used to fulfill the gap. Further, infrastructure abstraction extracts end-user applications from the node and network abstractions. Without any infrastructure abstraction, tailored solutions would be needed for each technology utilized by the application.

2.3. Infrastructure Abstraction. Infrastructure abstraction proposals typically describe the requirements with the same terms as network abstractions, and both are typically referred as WSN middleware. However, the functional units are different: on network abstraction, *heterogeneous nodes in one network* are abstracted behind an interface. On infrastructure abstraction, multiple *heterogeneous sensor networks* are abstracted behind one interface.

The main purpose of the infrastructure abstraction is to separate end-user application from the heterogeneous sensor networks. The infrastructure abstraction is a relatively new research area, which has gained more attention lately. This is due to the wide application space: new end-user applications are tested, deployed, and evaluated using different technologies. An infrastructure abstraction makes end-user application development faster, easier, and WSN technology independent. Currently, there is a lack of surveys on infrastructure abstractions, and the requirements and the design space have not been defined comprehensively. This paper is targeted at contributing in filling this knowledge cap.

3. Related Surveys

Mohamed and Al-Jaroodi [17] survey and discuss WSNs as service oriented middleware (SOM). A SOM provides WSNs as services for application developers. WSN derived requirements for SOMs are runtime for services, service discovery, heterogeneity abstraction, service configuration, service transparency, automated discovery and service change, interoperability between devices and systems, efficient handling of large volumes of data, security, and support for Quality of Service (QoS) [17]. However, Mohamed and Al-Jaroodi [17] state that very few of their surveyed work address even half of the presented requirements. We consider SOMs as a subset of infrastructure abstractions that mainly abstract network abstraction services from the application developer. For example, metadata or processing are not discussed in [17].

Dafei and Yu [18] survey sensor web proposals. By their definition, sensor web provides access, discovery, and interoperability of sensor services through WWW. This is one subset of infrastructure abstraction of the WSNs. For example, Yin et al. discuss context-awareness (referred to as processing in this paper) as an open issue in sensor webs. We consider that incorporating WWW as a part of the infrastructure abstraction is not a desired solution for every application, since connection to the Internet can be a security risk, for example, in factories and hospitals. Our survey concentrates on wider area of infrastructure abstractions.

Bröring et al. [19] discuss new generation sensor web enablement (SWE) of open geospatial consortium SWE (OGC). They present a layered stack for SWE, which consists of sensor layer, sensor web layer, and application layer. They identify four middleware classes (abstractions in this paper), which may overlap with each other and cross the three stack layers: sensor network management systems, sensor web infrastructure, centralized sensor web portals, and Internet of Things/Web of Things. Compared to our abstraction levels, their sensor network management systems class is the same as the node and the network abstraction levels. The rest of the classes belong to our infrastructure abstraction. Thus, Bröring et al. [19] present a more refined stack of the infrastructure abstraction, but this is done only for their definition of sensor webs. Further on, Bröring et al. [19] concentrate only to OGC SWE. Again, our survey concentrates on general infrastructure abstractions, and OGC SWE is one possible solution in this wide field.

4. Motivation

In our earlier work, we codeveloped WSN pilot applications with partners [20]: variety of interfaces were needed to integrate our WSN technology to the existing systems and tailoring was often needed for end-user application, be it weather web service, working conditions monitoring, factory automation system, or hospital personnel safety. These experiences motivated us to research abstractions for end-user application development to find proposals that remove tailoring and repeating work. We classified the abstractions to the presented three levels and studied their current status from existing surveys.

Although there are several proposals fitting to the infrastructure abstraction, only three related surveys were found, and they concentrated on one specific area of such abstractions. Also, related surveys suggested that infrastructure abstractions do not utilize processing capabilities of network abstractions, since it is not pointed by any of the three nor discussed. These findings motivated us to write this survey.

It is easy to envision several applications where a versatile infrastructure abstraction is a necessity. We describe a hospital use case as a motivating example: a fixed WSN is deployed for indoor positioning in a hospital. A patient can be equipped with a mobile device, such as a tablet, and another WSN that does physiological measurements, for example, patient temperature, electrocardiogram (ECG), and stress level. The tablet sends physiological and positioning data over WLAN or 3G to the infrastructure abstraction.

Patients can move around in the hospital, and they are tracked constantly. If the patient gets a seizure, an alarm of the location is sent to personnel. The ECG is stored for later study by a doctor. Further, the ECG is combined with location and indoor air condition information. These help the doctor to distinguish normal behavior from concerning situations. Finally, the patient can navigate in the hospital buildings with the help of the indoor positioning and the tablet. For example, if the patient is ordered to an X-ray and the tablet can lead him/her to the right place at the right time.

The presented hospital use case sets requirements for infrastructure abstractions: As patients check in and out, the infrastructure level processing must detect new patients and their measurement devices through *service discovery* in order to store, process, and monitor data related to the patients. *Technology interoperability* is an obvious necessity, since different technologies are used as data sources. *Data processing* is needed in several occasions: calculate position, combine position with patient data, detect abnormal ECG behavior, and so forth. Patient information and floor plans are examples of *metadata* required by the application.

5. Survey of the Infrastructure Abstraction

The material for this survey was collected using IEEE Xplorer and Google Scholar search engines. IEEE Xplorer returns 728 hits and Google Scholar approximately 15300 hits with *WSN* and *middleware* search words from year 2006 onwards. Only research papers abstracting WSNs for end-user application were considered for this survey. We selected a set of papers that emphasize the diversity of infrastructure abstractions and present comprehensive set of requirements. Further, we only concentrate on the high level designs and do not consider implementation specific details.

5.1. Open Geospatial Consortium Sensor Web Enablement. Open geospatial consortium (OGC) [21, 22] has proposed sensor web enablement (SWE), which is a set of XML specifications and interfaces for WSNs. OGC SWE has six specifications: sensor model language (SensorML) [23], observations and measurements (O&M) [24], sensor alert service (SAS) [25], sensor observation service (SOS) [26], sensor planning service (SPS) [27], and web notification service (WNS) [28].

SensorML is a set of models and XML schemas, which can be used for discovering services (including SensorML process models), tasking sensor services, processing observations (often measurements) with SensorML Process model, and chaining SensorML processes [21–23]. In addition, SensorML provides uniform data format for the OGC SWE services and contains metadata for processes. SensorML process has inputs, outputs, and parameters; a process without any inputs is a data source (e.g., a measurement device). O&M is a set of models and XML schema that describe the output information model for the sensor web applications [22, 24]. For example, an observation in O&M combines metadata, result, and sampling time in together. Now retired Transducer Markup Language (TML) defined a model for hardware characteristic of sensors and actuators, and a transportation method for sensor data [22, 29].

SOS is an interface to access observations with several parameters, such as temporal and geographical [22, 26]. It utilizes O&M for modeling sensor observations and SensorML for modeling sensors and sensor systems. SAS is an event stream processor and notification system [22, 25]. It continuously monitors data stream and creates events or alerts from pattern matching situations. SPS provides an interface to find out about available assets and possibility to execute tasks in the system [22, 27]. WNS is an interface for delivering notifications (events and alerts) to the user [22, 28]. Communication can be one-way, where notification is just delivered to the user, or two-way, where a response is expected from the user as well.

OGC SWE is an exhaustive but complete set of specifications for infrastructure abstraction. SensorML and O&M together resolve most of the abstraction problems. However, the OGC SWE does not distribute processing to the abstracted technologies; it only uses them as data providers. As mentioned earlier, not all WSN applications can be web services, although parts of the OGC SWE could be utilized without interaction in web: for example, SensorML and O&M could be used just as ontology. OGC SWE could be even considered too complex specification for some use-cases, for example, for resource constrained embedded devices performing simple interactions for actuating air conditioning according to temperature. Further, OGC SWE interoperability is restricted to sensor and actuator devices only and the metadata is mainly for describing physical features of the sensors and their measurements.

5.2. Bouillet et al. Bouillet et al. [30] present a middleware for creating sensor network applications that utilize data from many sources simultaneously. They reason it with application scenarios, which combine sensor data from temperature sensors to cameras and provide processed data for different end-users. The paper describes *Processing Elements* (PEs), which take data in as input streams, process data, and return result as one or more output streams. The middleware system discovers data source streams and connects them to correct PEs. Data sources and PEs are homogenized with Web Ontology Language (OWL) [31] ontologies. Further, PEs can be interconnected to refine processing more and to provide processed data for complex applications. Formal models are given for data sources and PEs. An algorithm is proposed, which can automatically composite applications from connecting PEs and data source streams according to a high level description of the application.

Bouillet et al. [30] proposal achieves technology interoperability with the OWL ontologies, if the connected technology can match the ontology format. The strongest part of the proposal is the processing: data from the sources can be refined and combined into infinity with connectible PEs. They state that in-network processing of data sources is transparent to the system, but their proposal does not seem to utilize this. The service discovery is limited to matching the input of a PE to data sources according to their semantic descriptions with the ontology: the temporal nature of the WSNs is not discussed, and therefore it is a question that can this proposal survive from, for example, disappearing data source. Metadata is not discussed in [30], but apparently metadata providers could be data sources and PEs could get their metadata through those.

5.3. Global Sensor Networks. Global Sensor Network (GSN) abstracts WSNs as a set of virtual sensors, which have one homogenized structure [32]. XML is used to define a virtual sensor, and each virtual sensor has one or multiple inputs consisting of any type of real sensors or other virtual sensors. SQL like language can be used to retrieve and process the data from the abstracted sensors through wrappers. A time model with count- and time-based windowing mechanism is presented to handle different application requirements for the temporal semantics. Metadata is used in these virtual sensors for service discovery and identification.

GSN is a complete proposal, which only lacks a definition of ontology. Instead, the data format is described as a part of the virtual sensor. This is a versatile approach, but the end-user application (or other virtual sensors) must understand all possible structures. If these are not defined globally, the application implementation can be cumbersome: for example, one virtual sensor could produce temperature as an integer and another as an double; the application should then figure out the differences. Further, GSN supports similar processing methods that network abstractions often provide, for example, averaging, but there is no indication that the virtual sensor abstraction would utilize these methods directly.

5.4. SenseWeb and SensorMap. SenseWeb collects wired, wireless and mobile sensors behind one application programming interface (API) to provide technology interoperability [33]. Coordinator forwards application requests for data, homogenizes data format, caches measurements to SenseDB, and provides service and resource discovery. Sensors are connected to the coordinator through tailored gateways, which uniform the access. DataHub gateway is provided as a reference gateway for those sensors that do not want to implement tailored one. A mobile proxy connects mobile sensors to SenseWeb and delivers measurements according to location when sensors are available. Data transformers are used to process data. Coordinator indexes transformers and provides transformer service discovery for the applications.

SensorMap [34] is an application on top of SenseWeb. It provides tools to illustrate sensor data on a map. It consists of GeoDB, DataHub, Aggregator, and SensorMap GUI. GeoDB holds metadata for sensors, DataHub keeps track of connected sensors, Aggregator combines geographically near-by sensors, and SensorMap GUI presents measurements on a map according to queries. SensorMap allows technology interoperability and service discovery through the GeoDB and DataHub. Data processing is provided by Aggregator and SensorMap GUI. Aggregator resolves nearby and similar sensors from the provided metadata. SensorMap fuses results of queries together with a map to visualize the data. Query results are queried from Aggregator and DataHub.

SenseWeb and SensorMap provide a complete way to aggregate data and present it on web. SenseWeb even realizes node mobility, which is often overlooked by the other proposals. Since SenseWeb and SensorMap work tightly in the Internet and WWW, they raise the question of security and privacy. However, the Internet and WWW approach is not a general solution for the infrastructure abstraction: stand-alone WSN applications are required as well. SenseWeb and SensorMap only collect web published sensor data together, for example, actuator controlling is not possible.

5.5. Lamses. Lamses is a large-scale middleware proposal for ubiquitous sensor networks [35] with a complex architecture that concentrates on creating context-aware applications from sensor network data. Lamses provides a common interface for accessing abstracted WSNs technologies and managing applications. Lamses consists of a context aware engine, metainformation management, sensor network management, control and query management, and state management. The context-aware engine processes sensor data and events to create context-aware events. The data is handled and stored as XML packets, which are provided as queries for the application programs as well.

Lamses supports only context-aware processing, although it internally integrates data for the context-aware engine. It has a control and query management, which could deliver data or controlling commands to the abstracted sensor networks, but this is not discussed in the paper. The metadata describes only a limited set of attributes of the WSN devices mainly related to the sensing hardware and device identifiers.

5.6. SeNsIM. SeNsIM proposes an architectural and a data model for technology interoperability between sensing technologies [36]. It adapts existing technologies with wrappers and provides a mediator interface for end-user applications. The wrappers connect to the mediator, and the mediator uses an XML query interface for end-user applications. The data is unified by formatting it to an XML. SeNsIM does not provide any metadata or processing support. Casola et al. [36] implicate that processing of the abstracted technologies could be utilized, and that "the state of the sensor can be modified," which indicates that SeNsIM could deliver data to the abstracted technologies. However, these are not discussed clearly in the paper. The wrappers do service discovery on to the abstracted technologies, but it is not clarified how mediator shows this to the applications.

5.7. Smart-M3. Smart-M3 [37] is an interoperability platform for smart spaces. It allows small embedded devices to locally share semantic information. Any ontology can be used with it, and application developer can use ontology through Ontology API. Every device, or *Knowledge Processor* (KP), can store and retrieve information from the *Semantic Information Broker* (SIB). For example, a mobile phone can be used to control local sensor network actuators and home appliances through Smart-M3. KPs can only communicate through a SIB by inserting, querying, or subscribing/publishing the data into it. KPs can be mobile: they can join and leave to a SIB and they can discover the data of other KPs from it.

Smart-M3 is more a technology interoperation communication protocol than a complete infrastructure abstraction, but it does not propose any restrictions for the application development. However, it does require a common ontology for information storing. Without a standardized ontology, Smart-M3 will be only locally usable. For example, Smart-M3 implementation in home applications can use different ontology from Smart-M3 implementation in office applications. Switching between these locations with the same mobile device will require implementation of both ontologies (or use of both Ontology APIs) on that mobile device. Smart-M3 does not provide metadata or processing support. If these are required, they must be implemented on top of Smart-M3.

Smart-M3 has an unique approach compared to other surveyed proposals that each embedded device can directly interoperate through it: there is no need for end-user application or processing run-time in the infrastructure; the actuator device controlling the air condition can read by itself from the SIB what carbon dioxide and temperature sensors have reported and adjust the air according to those values.

6. Infrastructure Abstraction Requirements

The common features of the surveyed work are gathered in Table 1. These requirements are ruled by the end-user applications, and therefore are the requirements for the infrastructure abstraction. It should be noted that the basic paradigm of the infrastructure abstraction is to separate the end-user application from the abstracted technologies: the technologies behind the infrastructure abstraction can change without any need to modify existing end-user applications. The requirements are discussed in detail in the following and the references given in this Section are for example of the discussed topic.

6.1. Technology Interoperability. Heterogeneous WSN technologies must be homogenized for the end-user application. This is one of the major challenges with infrastructure

Table 1: Features of the surveyed infrastructure abstractions.

Abstraction	Data access method	Technology interoperability	Ontology	Service discovery	Metadata	Processing
OGC SWE	Publish/Subscribe, queried streams and historical values, events, and alerts	A set of XML specifications and interfaces for sensor interoperability	Ontology not mentioned, but SensorML describes uniform data structure and units, and O&M describes observation structure	SensorML models and XML schema can be mined for SensorML processes (includes connected sensors and actuators); SPS can be used to test asset availability for a task	SensorML contains process metadata and O&M contain observation metadata	Through SensorML processes and process chains; SAS for creating notifications from pattern matching data
Bouillet et al. [30]	Stream, but the accessing method is not described	Through the input semantics that data sources must adapt	OWL ontologies	Data sources and PE outputs that match to a PE input can be discovered	Not mentioned	PEs can be interconnected; an algorithm to combine PEs into an application
GNS	Query with streams and historical data	Abstracted technologies are described as virtual sensors	No ontology, but the virtual sensor describes the data structure freely	Virtual sensor can be discovered with their metadata descriptions	Each virtual sensor holds metadata descriptions	SQL like processing on the virtual sensors; virtual sensors can be sources to other virtual sensors
SenseWeb	Query with streams and historical data	Uniform API, with data and time abstraction	Not used, but the API homogenizes data format, but the method is not described	For sensors, according to sensor type or location, and for transformers	Not mentioned	Transformers can convert units, aggregate, fuse, and visualize data
SensorMap	Query with streams and historical data; location-based queries possible	Through SenseWeb DataHub	A lack of standard ontology discussed	Services can be discovered from GeoDB meta-data	GeoDB holds meta-data	Aggregate geographically close sensors and display on a map
Lamses	Events for context-awareness and queries for data retrieval	Describes a common interface for attaching WSNs into it	Not mentioned, but uses XML for homogenizing data	Not mentioned	An XML based meta-information database for complementary hardware information	Context-aware event and data processing; internally integrates data for context-aware events
SeNsIM	Query for real-time data and events	Wrappers connect technologies to a mediator and data is unified in an XML	Not mentioned, but an XML model is used for the data unifying	Wrappers discover sensors from abstracted technologies	Not mentioned.	Not mentioned clearly
Smart-M3	Publish/Subscribe and query/insert data	KPs can communicate through a SIB in a smart space without temporal connections	Any ontology can be used and there is an Ontology API for each used ontology	KPs can find different services through a SIB, if the semantics of their data match	Not mentioned	Not supported/possible

abstractions. Technology interoperability consists of three requirements: uniform data access, ontology, and technology feature homogenization.

6.1.1. Uniform Data Access. There must be a method to access the data of the abstracted technologies. Data retrieval can be history exploration [32], continuous stream [30], or event based [21]. On history exploration, the application requests existing data within a time window from the abstraction. On continuous stream, data is delivered continuously to the application. On event based, data is delivered after certain event has occurred either continuously or as a one-time action. Events are often referred as alerts in many publications.

A query or publish/subscribe interface is typically used for accessing the data. On query interface, the application retrieves information with explicit queries [32]. On publish/subscribe interface [21, 37], the application subscribes for interesting data and the publisher delivers the data when it is available. Queries fit well for history exploration and continuous stream retrievals, whereas publish/subscribe fits for continuous stream and event retrievals.

6.1.2. Ontology for Data Format Homogenization. The main task for ontology is to remove heterogeneity between different data producing technologies for the same data type [33]. Ontology describes format, units, and ranges for the data. This simplifies end-user application development, for example, application can rely that temperature measurement is always in the same format and has a unit of Celsius. If data would be requested from different sources without ontology, end-user applications would have to parse and format the data from each technology separately for the final presentation. With a common ontology, the end-user application becomes technology independent: underlying technologies can be changed as long as they can produce data in the ontology format.

6.1.3. Technology Feature Homogenization. WSN technologies have several features, which should be homogenized. The list of such features could be exhaustive and in some case, transparency is a necessity. For example, end-user should know that the node controlling the power outlet of TV is that particular one he/she sees on the wall. Configuration, concept of time [32], and data delivery back to the abstracted technologies are common features that typically need homogenization at the infrastructure abstraction.

Resource constrained WSNs seldom have a real-time clock and/or the data packets are limited in size and cannot deliver the exact time of the observed phenomena. In some applications, it is vital to understand temporal connections between observed phenomenas [32], even if they are observed with different technologies. Therefore, the infrastructure abstraction should homogenize time format. This is an obvious part of the ontology.

WSNs and other supporting technologies are not only data providers, but are also consumers. There must be a method to deliver data to the abstracted technologies. For example, WSN can deliver measurement data, which is then used to control actuated device, such as air conditioning. Typically, this functionality is part of the data access. In addition to control tasks, there can be configuration tasks and data injection that abstraction should support.

6.2. Service Discovery. Not all services are continuously accessible on WSNs due to the possible node mobility, error-prone communication medium, hardware failures, or energy depletion. In addition, new services can be added to the WSN or the supporting technologies. For example, in web services such as SensorMap [34], third parties can add or remove data providers, which then appear as usable data sources for the already existing end-user applications. A service discovery method is needed to find the interesting services [32, 33].

The infrastructure abstraction must solve three main tasks in service discovery. First, networks may have their own service discovery methods, which must be homogenized to ease the end-user application development. Second, the service discovery must be general enough to expand discovery to those technologies/services that wither do not follow the existing WSN paradigm or are completely different from typical measurement services. Third, scalability and transparency are needed. For example, most applications are not interested about the sensor itself, only the measurement type. However, the accuracy of the sensor may be important on some applications and even the sensor product name and the manufacturer information may be needed. If these are used as service discovery parameters, the service discovery must scale to various levels of detail and be transparent when needed.

6.3. Metadata. Plain measurement data is not sufficient for the end-user applications but it must be completed with various kinds of metadata. Descriptions of location, hardware, measurement accuracy, measurement purpose, and so forth, are typical metadata for WSN measurements [35]. For example, location description could be "kitchen" or an identifier for a patient in patient monitoring. Even more complex metadata is a possible requirement. For example, if the measurement is made on a certain geographical location, the end-user might expect to see the measurement illustrated on a map, and the location must be updated, when the measurement is location information, or the node is mobile [34].

Metadata is such a wide topic that an infrastructure abstraction cannot specify all the possible metadata that an end-user application might require. However, the infrastructure abstraction should support at least the metadata that abstracted technologies typically provide. If there is heterogeneity in the available metadata between different abstracted technologies, the infrastructure abstraction should allow adding same metadata for all the abstracted technologies. For example, the infrastructure abstraction could require from the technology adapter that it completes required set of metadata to the measurements. Some of the metadata can be provided as a service as well. For example, a map repository could be a service of the infrastructure abstraction.

6.4. Processing. A method to create and add processing services to the infrastructure abstraction produce at least two benefits. (i) Aggregated data can be reused between different end-user applications or between different instances of the end-user application. This will reduce data requests to the network, reduce data traffic, and make application development simpler, when aggregation services are needed (e.g., averaging or summing). (ii) Infrastructure has more resources for processing than resource constrained WSNs. Thus, it is possible to do more complex aggregation and processing on the infrastructure-level. The infrastructure abstraction can then provide more elegant services to the end-user application, if needed. For example, infrastructure-level processing could distinguish interesting situations from the measurement data and create an event or new measurement for the end-user applications.

The infrastructure abstraction should allow creating new processing services [30, 32]. In addition to the already presented service access requirements, processing services require an execution environment, a method to describe/create the processing services, and an injection method to add new processing services.

7. Discussion and Open Research Questions

The open research problems yield from the wide application space, which requires infrastructure abstraction to be versatile. Currently, many existing solutions are restricted to one application field or on web-based applications. Although each of them eases the application development, a general purpose infrastructure abstraction would yield easier adaptation, since developers can start to develop on the same abstraction without burdensome pre-examination of the WSN technologies. WSNs need standardizations and a commonly used abstraction in order to finally breakthrough. We consider that there is still research work and problems to be solved.

Security is discussed in only few of the surveyed papers. Mohamed and Al-Jaroodi [17] found this same issue with their survey, and debated about the importance of security in SOMs. They consider security via communication and operation safety and conclude that security is indeed an important topic. However, we consider that at the infrastructure level, communication or operation security is not a research problem as such, since the existing solutions are sufficient. For example, most infrastructure abstractions are traditional server or web software, which can rely on SSL, SSH, or other widely used schemes. The WSN has its own security mechanisms, which are not exposed to the upper levels. More prominent security topics are user access rights (e.g., who can access what data, and from where), processing provisioning (e.g., what is dropped, if the system processing capabilities saturate), or accounting (e.g., who has consumed data, and by how much).

Quality of Service (QoS) is typically a network level problem, and the topic is not discussed in the surveyed papers. Typically, there is no QoS support from the sensor network or it is already abstracted by the network abstraction, for example, TinyDB does not provide clear QoS support for the application. However, the infrastructure abstraction should take care of using the QoS support if such is available. The infrastructure abstraction should classify the application required data and demand better quality for the important data from the abstracted sensor networks to ensure the fastest and the most reliable data delivery.

QoS in the infrastructure abstraction is another open question. Infrastructure may need to receive and process significant amount of data, and still react to alarming conditions quickly. This problem can be seen as implementation or server infrastructure problem, but novel location, data, and/or user aware distribution approaches of the infrastructure may be needed.

For the ontology, the most demanding open task is to develop one that is accepted as a de facto standard. Currently, there is no such proposal. The problem is the versatility: ontology should support all the possible data sources that are required or will be introduced in future. This is not an easy task and will require novel approach or may well be impossible. Defining or modeling the WSN application space as well as possible is the ground work that should be made before such ontology can be produced.

Service discovery is incorporated on most of the surveyed proposals. Often papers only state that some feature of the proposal can be used for service discovery without any further analyzing the usability. Service discovery should gain more attention, since it is not a trivial task. The homogenization, generalization, scalability, and transparency features should be fulfilled.

Processing is supported by many of the surveyed abstractions, but there is lack of support for automated feedback loops in many proposals. WSNs are often used in controlling applications and if the infrastructure-level processing is versatile enough, the processing task can even make the controlling decision. This requires a feedback loop from the processing to the abstracted technologies. This should not be overlooked when designing an infrastructure abstraction.

Many network abstractions support in-network processing. The infrastructure abstraction should utilize these features, since there are clear benefits available from distributing the processing to the network where applicable. Krishnamachari et al. [38] found out with mathematical and simulation analysis of data aggregation models that in-network aggregation can produce 50%–80% energy savings in WSNs. Prakash et al. [39] simulated models with real hardware parameters and found out that simple in-network averaging can reduce energy consumption from 260–270 mJ to 60–70 mJ. Luo et al. [40] evaluated in-network aggregating queries for TinyDB [11] with TAG [41], and SKETCH [42]. They found out that in-network aggregation can reduce energy consumption, improve data quality, and/or reduce end-to-end packet loss rate. Currently, distributing processing in the abstracted networks is not implemented in any of the surveyed proposals.

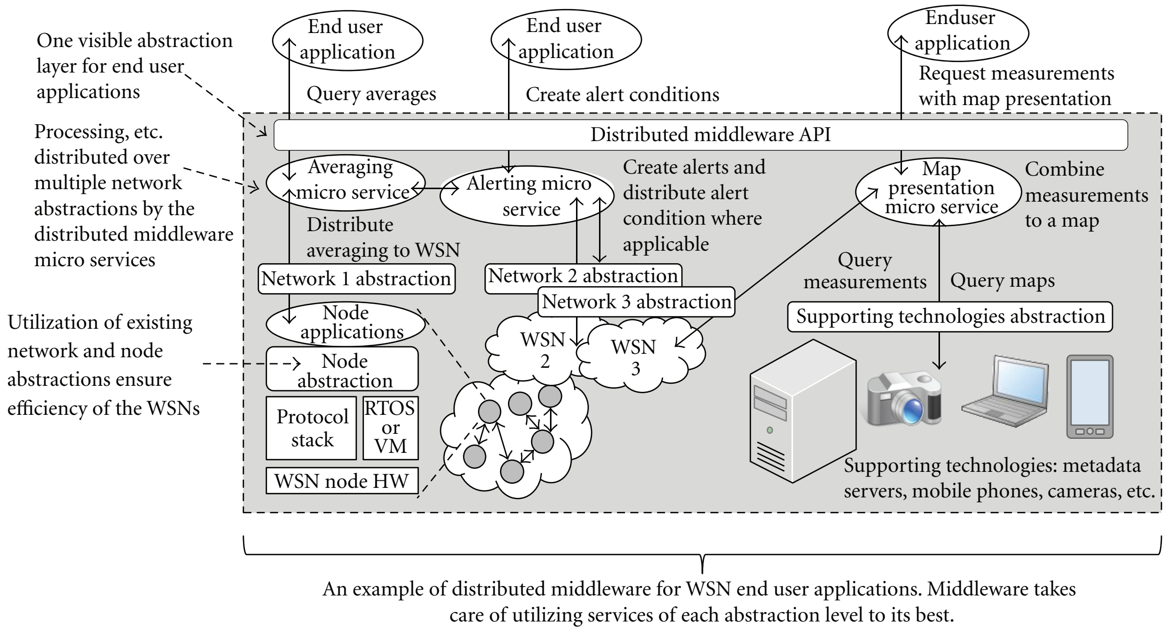

End user application
End user application
Enduser application
One visible abstraction layer for end user applications
Query averages
Create alert conditions
Request measurements with map presentation
Distributed middleware API
Processing, etc. distributed over multiple network abstractions by the distributed middleware micro services
Averaging micro service
Alerting micro service
Create alerts and distribute alert condition where applicable
Map presentation micro service
Combine measurements to a map
Distribute averaging to WSN
Query measurements
Query maps
Network 1 abstraction
Network 2 abstraction
Network 3 abstraction
Supporting technologies abstraction
Node applications
WSN 2
WSN 3
Utilization of existing network and node abstractions ensure efficiency of the WSNs
Node abstraction
Protocol stack
RTOS or VM
WSN node HW
Supporting technologies: metadata servers, mobile phones, cameras, etc.
An example of distributed middleware for WSN end user applications. Middleware takes care of utilizing services of each abstraction level to its best.

FIGURE 2: An example use case of distributed middleware design. Averaging micro service distributes the average calculation to the Network 1.

It should be noted that both Prakash et al. [39] and Luo et al. [40] found out energy consumption overhead of in-network processing as well, when the processing is complex or distributed over large areas of sparse network. Therefore, it should be carefully selected where the processing is executed: on the network or on the infrastructure. As far as the authors of this paper know, there is no existing proposal for algorithms or models to distinguish the best place for the processing to take place. This is an obvious open research problem. With careful distribution of the processing to the network, WSN performance can be improved energy and reliability wise.

From a scientific point of view, there are no metrics, benchmarks, or testing facilities for comparing infrastructure abstractions. These can be created, for example, the size of the data structures in different proposals could be compared using typical measurements, or the saturation point of the processing capabilities could be tested. As far as the authors know, such metrics have not been published or tested for infrastructure abstractions.

8. Design Proposals for Open Research Questions

We propose distributed middleware as a common naming convention for WSN abstraction that work on node, network, and infrastructure levels. Figure 2 presents how the distributed middleware fits to the WSN abstraction levels presented in Figure 1. We have concentrated to solve open problems in the service discovery, the distributed processing and the ontology. The key idea is to use small processing services that are simple enough to be distributed through the network abstraction, if it supports such action. If the distribution is not possible, the infrastructure handles the processing.

8.1. Tag-Based Service Discovery. For service discovery we propose categorized tags. For example, *location: Tampere* has a category *location* and a tag *Tampere*. An application can make a query of tags to discover existing and new services. The tag-based discovery is scalable, transparent, and future-proof, since it does not restrict the tags. For example, the service can have a tag that describes the sensor model *sensor: dallas*; *semiconductor*; *dm620*, and the same service can also complete tag search of *measurement*: *temperature*. Further, new technologies can always introduce tags that have not been used before, if they are needed with new applications. GSN has quite similar method to present metadata for discovering the virtual sensors.

Tags can be generated from the existing metadata, and they work as part of the metadata as well. For example, if the location of the sensor is known as coordinates, they can be converted to descriptive tags of the location. Further, tags can be easily generated from the data that sensor provides. If there is a temperature measurement value retrieved from the sensor, it should respond to tag query of *measurement*: *temperature*.

8.2. Distributed Processing. Distributed middleware design requires specifying interface between network and infrastructure abstractions, and creating a processing language that can be analyzed to find processing tasks that can be distributed to the network. We have started to design an XML specification, which describes the processing task called *micro service*. The typical in-network aggregating/processing operations are key components in describing these micro

```
<microservice>
  <descriptions><!- -defines what the service does and describes tags
                 and address, which can be used to find it- -></description>
  <input>
    <source name="src1"><!- -name defines name for this source in the processing part- ->
      <address><!- -this is direct access to the service, thus we already know it- -></address>
    </source>
    <source name="src2">
      <discovery><!- -Search for all outdoor temperature values of Tampere- ->
      <tag>location : tampere</tag>
      <tag>location : outdoor</tag>
      <tag>measurement : temperature</tag>
      </discovery>
    </source>
  </input>
  <output><!- -defines output(s) for this service- -></output>
  <process>
    <variable name="avg">
      <assign>
        <function type="average">
          <parameter>src1</parameter>
          <parameter>src2</parameter>
        </function>
      </assign>
    </variable>
    <if ><compare operator="greater"><valueOf name="avg"/>25.0 </compare>
    <then>
      <action type="increase"><destination name="dst"/>
    </then>
    </if>
    <if><compare operator="less"><valueOf name="avg"/>20.0</compare>
    <then>
      <action type="decrease"><destination name="dst"/>
    </then>
    </if>
  </process>
</microservice>
```

Algorithm 1: An example of the processing section of the micro service XML schema.

services. When the input technology supports used aggregation method, the infrastructure abstraction execution environment distributes processing through the network abstraction.

As an example, a pseudo-XML schema is proposed to describe a new micro service in Algorithm 1. First, the micro service is described. This includes the tags and address, which are used to find the micro service and to connect it. Second, the input sources are defined. The input can be either direct address to a service or a set of discovery tags. Third, the output is defined. The output is either direct connection to another micro service (e.g., for the feedback control loops) or a server that responds to the application queries. Finally, the processing of the micro service is described. The inputs are processed according to the processing schema and the results are driven to the output. The presented processing calculates average between two sources. If these sources can calculate average in-network, the processing is distributed through the network abstraction, for example, if the "src1" and "src2" are in the same network behind TinyDB network abstraction, the averaging could be executed by it. If there is no support or sources are from different networks, the infrastructure processing calculates the average.

8.3. Ontology. Instead of trying to create omnipotent ontology as a de facto standard, world could be squeezed into the mold that ontology provides. For example, world could be seen as measurements, where everything can be a measurement or can be measured. Expanding this vision to nodes and networks allows covering all kind of data in WSN applications with simple ontology structure. For example, [19] discusses integration of social networks as a part of sensor web as an open issue. As an example, Facebook type of social network could be expressed with the sensor centric ontology of WSN OpenAPI [43]: Facebook user's friends are a network, where every node is a friend and their status, birthday, pictures, and so forth, are measurements. Further, a person can be a network of measurements, such as body temperature, first name, and social security number. A car is a network of sensors, and so forth.

9. Conclusions

This paper classified three levels of abstraction for WSNs: node, network, and infrastructure abstractions. A survey of existing infrastructure abstractions was presented and infrastructure abstraction requirements were defined. An infrastructure abstraction should provide technology interoperability by homogenizing data access, data format, and other technology features. Service discovery, metadata addition, and data processing are also required. As open research questions we discussed security and QoS and defined lack of de facto ontology, lack of focus on service discovery, lack of processing distribution to the network, and lack of benchmark metrics for infrastructure abstractions. We discussed a solution for the ontology problem and proposed design directions for the service discovery and distributed processing. As a future work, we will implement proposed designs and test them with simulations and real world deployments. Also, we will study benchmarking metrics for infrastructure abstractions.

References

[1] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci, "Wireless sensor networks: a survey," *Computer Networks*, vol. 38, no. 4, pp. 393–422, 2002.

[2] R. Sugihara and R. K. Gupta, "Programming models for sensor networks: a survey," *ACM Transactions on Sensor Networks*, vol. 4, no. 2, article no. 8, 2008.

[3] L. Mottola and G. P. Picco, "Programming wireless sensor networks: fundamental concepts and state of the art," *ACM Computing Surveys*, vol. 43, no. 3, article no. 19, 2011.

[4] K. Henricksen and R. Robinson, "A survey of middleware for sensor networks: State-of-the-art and future directions," in *Proceedings of the International Workshop on Middleware for Sensor Networks (MidSens '06)*, pp. 60–65, New York, NY, USA, November 2006.

[5] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Notices*, vol. 35, no. 11, pp. 93–104, 2000.

[6] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki—a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*, pp. 455–462, Tampa, Fla, USA, November 2004.

[7] P. Levis and D. Culler, "Mate: a tiny virtual machine for sensor networks," *SIGPLAN Notices*, vol. 37, no. 10, pp. 85–95, 2002.

[8] D. Gay, P. Levis, R. Von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," *SIGPLAN Notices*, vol. 38, no. 5, pp. 1–11, 2003.

[9] S. Hadim and N. Mohamed, "Middleware: Middleware challenges and approaches for wireless sensor networks," *IEEE Distributed Systems Online*, vol. 7, no. 3, pp. 1–23, 2006.

[10] M. M. Wang, J. N. Cao, J. Li, and S. K. Dasi, "Middleware for wireless sensor networks: a survey," *Journal of Computer Science and Technology*, vol. 23, no. 3, pp. 305–326, 2008.

[11] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[12] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *SIGMOD Record*, vol. 31, no. 3, pp. 9–18, 2002.

[13] C. L. Fok, G. C. Roman, and C. Lu, "Mobile agent middleware for sensor networks: an application case study," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*, pp. 382–387, April 2005.

[14] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "TinyLIME: Bridging mobile and sensor networks through middleware," in *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom '05)*, pp. 61–74, March 2005.

[15] M. M. Molla and S. I. Ahamed, "A survey of middleware for sensor network and challenges," in *Proceedings of the International Conference on Parallel Processing Workshops (ICPP '06)*, pp. 223–228, August 2006.

[16] W. Masri and Z. Mammen, "Middleware for wireless sensor networks: a comparative analysis," in *Proceedings of the IFIP International Conference on Network and Parallel Computing Workshops (NPC '07)*, pp. 349–356, September 2007.

[17] N. Mohamed and J. Al-Jaroodi, "Service-oriented middleware approaches for wireless sensor networks," in *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS-44 '10)*, pp. 1–9, January 2011.

[18] Y. Dafei and F. Yu, "From sensor net to sensor grid: a survey and taxonomy on Sensor Web," in *Proceedings of the IEEE International Geoscience and Remote Sensing Symposium (IGARSS '07)*, pp. 2935–2938, June 2007.

[19] A. Bröring, J. Echterhoff, S. Jirka et al., "New generation sensor web enablement," *Sensors*, vol. 11, no. 3, pp. 2652–2699, 2011.

[20] T. Laukkarinen, J. Suhonen, T. D. Hamalainen, and M. Hannikainen, "Pilot studies of wireless sensor networks: Practical experiences," in *Proceedings of the Conference on Design and Architectures for Signal and Image Processing (DASIP '11)*, pp. 1–18, November 2011.

[21] M. Botts, G. Percivall, C. Reed, and J. Davidson, "Ogc sensor web enablement: overview and high level architecture," in *GeoSensor Networks*, S. Nittel, A. Labrinidis, and A. Stefanidis, Eds., vol. 4540 of *Lecture Notes in Computer Science*, pp. 175–190, Springer, Berlin, UK, 2008.

[22] I. Simonis, "Ogc sensor web enablement architecture, version 0.4.0," OGC 06-021r4, OGC Best Practice. Open Geospatial Consortium, 2008.

[23] M. Botts and A. Robin, "Ogc sensor model language (sensorml) implementation specification, version 1.0.0, OpenGIS Implementation Specification," OGC 07-000, Open Geospatial Consortium, 2007.

[24] S. Cox, "Observations and measurements—xml implementation, version 2.0, OpenGIS Implementation standard," OGC 10-025r1, Open Geospatial Consortium, 2011.

[25] I. Simonis, "Ogc sensor alert service candidate implementation specification, version 0.9," OGC 06-028r3, Candidate OpenGIS Implementation Specification Open Geospatial Consortium, 2006.

[26] A. Bröring, C. Stasch, and J. Echterhoff, "Ogc sensor observation service interface standard, version 2.0," OGC 12-006, OpenGIS Implementation Standard. Open Geospatial Consortium, 2012.

[27] I. Simonis and J. Echterhoff, "Ogc sensor planning service implementation standard, version 2.0," OGC 09-000,

OpenGIS Implementation Standard. Open Geospatial Consortium, 2011.

[28] I. Simonis and A. Wytzisk, "Web notification service, version 0.1.0," OGC 03-008r2, OpenGIS Discussion Paper. Open Geospatial Consortium, 2003.

[29] S. Havens, "Ogc transducer markup language (tml) implementation specification, version 1.0.0, retired," OGC 06-010r6, OpenGIS Implementation Specification. Open Geospatial Consortium, 2007.

[30] E. Bouillet, M. Feblowitz, Z. Liu, A. Ranganathan, A. Riabov, and F. Ye, "A semantics-based middleware for utilizing heterogeneous sensor networks," *Distributed Computing in Sensor Systems*, vol. 4549, pp. 174–188, 2007.

[31] W3C, "Owl web ontology language reference," http://www.w3.org/TR/owl-ref, 2012.

[32] K. Aberer, M. Hauswirth, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks," in *Proceedings of the 8th International Conference on Mobile Data Management (MDM '07)*, pp. 198–205, May 2007.

[33] A. Kansal, S. Nath, J. Liu, and F. Zhao, "SenseWeb: an infrastructure for shared sensing," *IEEE Multimedia*, vol. 14, no. 4, pp. 8–13, 2007.

[34] S. Nath, J. Liu, and F. Zhao, "Challenges in building a portal for sensors world-wide," in *Proceedings of the 1st Workshop on WorldSensor-Web*, ACM, pp. 3–4, Boulder, 2006.

[35] Y. S. Jeong, E. H. Song, G. B. Chae, M. Hong, and D. S. Park, "Large-scale middleware for ubiquitous sensor networks," *IEEE Intelligent Systems*, vol. 25, no. 2, pp. 48–59, 2010.

[36] V. Casola, A. Gaglione, and A. Mazzeo, "A reference architecture for sensor networks integration and management," *GeoSensor Networks*, vol. 5659, pp. 158–168, 2009.

[37] J. Honkola, H. Laine, R. Brown, and O. Tyrkkö, "Smart-M3 information sharing platform," in *Proceedings of the 15th IEEE Symposium on Computers and Communications (ISCC '10)*, pp. 1041–1046, June 2010.

[38] L. Krishnamachari, D. Estrin, and S. Wicker, "The impact of data aggregation in wireless sensor networks," in *Proceedings of the 22nd International Conference on Distributed Computing Systems Workshops*, pp. 575–5578, 2002.

[39] G. L. Prakash, M. Thejaswini, S. H. Manjula, K. R. Venugopal, and L. M. Patnaik, "Energy efficient in-network data processing in sensor networks," *World Academy of Science, Engineering and Technology*, vol. 48, 2008.

[40] Q. Luo, H. Wu, W. Xue, and B. He, "Benchmarking in-network sensor query processing," Tech. Rep., The Hong Kong University of Science and Technology, 2005.

[41] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tag: a tiny aggregation service for ad-hoc sensor networks," in *Proceedings of the 5th symposium on Operating systems design and implementation, ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 131–146, 2002.

[42] J. Considine, F. Li, G. Kollios, and J. Byers, "Approximate aggregation techniques for sensor databases," in *Proceedings of the 20th International Conference on Data Engineering (ICDE '04)*, pp. 449–460, April 2004.

[43] J. Suhonen, O. Kivela, T. Laukkarinen, and M. Hannikainen, "Unified service access for wireless sensor networks," in *Proceedings of the 3rd International Workshop on Software Engineering for Sensor Network Applications (SESENA '12)*, pp. 49–495, June 2012.

PUBLICATION 5

J. Suhonen, O. Kivela, T. Laukkarinen, and M. Hännikäinen, "Unified service access for wireless sensor networks," *Third International Workshop on Software Engineering for Sensor Network Applications (SESENA)*, pp. 49–55, June 2, 2012. doi:10.1109/SESENA.2012.6225735

PUBLICATION 6

T. Laukkarinen, J. Suhonen, and M. Hännikäinen, “An embedded cloud design for Internet-of-Things,” *International Journal of Distributed Sensor Networks*, vol. 2013, Article ID 790130, 13 pages, 2013. doi:10.1155/2013/790130

First published by Hindawi Publishing Corporation.

Hindawi Publishing Corporation
International Journal of Distributed Sensor Networks
Volume 2013, Article ID 790130, 13 pages
http://dx.doi.org/10.1155/2013/790130

Research Article

An Embedded Cloud Design for Internet-of-Things

Teemu Laukkarinen, Jukka Suhonen, and Marko Hännikäinen

Department of Pervasive Computing, Tampere University of Technology, P.O. Box 553, 33101 Tampere, Finland

Correspondence should be addressed to Teemu Laukkarinen; teemu.laukkarinen@tut.fi

Received 11 April 2013; Revised 23 October 2013; Accepted 28 October 2013

Academic Editor: Seokcheon Lee

Internet-of-Things (IoT) consists of interconnected heterogeneous devices that ubiquitously interact with physical world. The devices are often resource constrained in terms of energy, computation, and communication resources. Distributing processing between these heterogeneous devices could yield to better performance and sharing, and extending resources of the devices could yield to more intelligent ubiquitous applications. Such a design can be called as "embedded cloud", which is defined in this paper. An embedded cloud design is presented that consists of distributable Process Description Language (PDL), Distributed Middleware (DiMiWa), and an infrastructure. As a result, PDL can execute distributed processes and share resources as services over heterogeneous IoT devices with help of DiMiWa and the infrastructure. The design is evaluated with a prototype implementation, where PDL and DiMiWa are executed on a small 8-bit microcontroller-based IoT device. The implementation requires only 5122 B of program memory (4% of the available), consumes under 1 ms of CPU time per process in the worst case, and allows over 100 simultaneous services per device.

1. Introduction

Internet-of-Things (IoT) consists of heterogeneous networked embedded devices that communicate through wired or wireless links or the Internet to create new ubiquitous, mash up, context aware, and information-based applications [1–3]. Often these embedded devices measure or interact with physical world ubiquitously, and they consist of wired sensors, RFIDs, Wireless Sensor Networks (WSNs), and/or mobile devices. Typically, IoT devices have limited communication and processing capabilities due to the energy consuming communication, small physical factor, battery operation, and long lifetime expectations.

Distributing and networking tens or hundreds of IoT devices enable intelligent ubiquitous applications. Distributing the processing or the application logic over the networked IoT devices is an important feature [3], since in-network processing can improve energy efficiency and data delivery reliability due to the reduced communication and congestion, especially, on resource-constrained WSNs [4–7]. Current IoT abstractions that hide device heterogeneity from end-user application development do not take distributed or in-network processing into account. The processing is done in the infrastructure, and the IoT devices are mainly used as heterogeneous data providers that are homogenized for the end-user applications [3, 4, 8]. The heterogeneity of the IoT devices makes distributing processing difficult, which may be the reason for the lack of such proposals. In this paper, heterogeneity means wired and wireless measuring and actuating technologies that use different communication methods and data formats without any direct device-to-device interoperation compatibility.

Modern mobile phones already use cloud services to extend their resources, such as iCloud on Apple devices and SkyDrive on Microsoft devices. On current IoT abstractions, IoT technologies work only as data providers and the application is implemented on the infrastructure [4]. However, the heterogeneous IoT devices in one location could potentially implement the application on their own, if they could expand and share their resources. As found out by Parwekar [3] and the authors of this paper [4], there is a lack of design proposals for IoT devices that would allow distributing processing and expanding resources between different IoT technologies.

We propose embedded cloud as a solution that would share, distribute, and expand resources of heterogeneous IoT technologies. As a contribution, we define the requirements for an embedded cloud and propose an embedded cloud design, which extends resources of measuring and actuating IoT devices. The novelties of the presented design are as follow.

(i) Heterogeneous IoT technologies can access external resources (other IoT technologies services and cloud processing) from the embedded cloud and virtually extend their own resources to the cloud services through the presented Distributed Middleware (DiMiWa).

(ii) The application logic is developed with a device independent Process Description Language (PDL), which is simple enough to be implemented and executed even on small 8-bit resource-constrained WSN devices, and versatile enough to allow implementing complex processes with a small memory footprint.

(iii) The application logic can be executed on the IoT devices of the suitable parts; there is no need to route everything through central arbitrating server. Further, technology specific processing can be harnessed, which allows to achieve the energy conservation benefits of in-network processing [4–7].

(iv) The design does not discriminate any technology; a solution to connect any measuring or actuating device as a part of the proposed embedded cloud design is presented.

PDL allows distributable process execution and a simple platform independent process creation with a small memory footprint. DiMiWa abstracts technology heterogeneity into services and provides sharing and expanding of the services (both *local* and *remote* services without visible separation to the application) while ensuring efficient use of the IoT technologies. The infrastructure provides PDL and DiMiWa execution environment for those technologies that cannot match the running requirements. Also, the infrastructure works as a communication arbiter and a data storage and provides those processing tasks that are too resource consuming for the connected IoT devices. Together these three components form an embedded cloud. The design is evaluated by studying the feasibility with a prototype implementation, three use cases, a scalability evaluation, and a comparison of the features against the related research.

The paper is constructed as follows. Section 2 gives our definition and requirements for an embedded cloud. Section 3 covers the related work of IoT clouds and the components of the presented embedded cloud design. The embedded cloud design is given in Section 4, and it is evaluated with a prototype implementation in Section 5. Section 6 discusses open questions in the design. Finally, Section 7 concludes the paper and gives future work.

2. The Definition of the Embedded Cloud

Defining cloud computing has been a cumbersome task and no widely accepted definition can be found [9, 10]. Infrastructure, platform, and software as a service (IaaS, PaaS, and SaaS) are the main cloud computing approaches, which are results of service oriented architecture (SOA). At the time of writing, a Google search of the "embedded cloud" gives approx. 25000 hits, and the first hits redefine "cloud" of infrastructure connected IoT devices as an embedded cloud [11–13]. A Google Scholar search gives approx. 248 hits for the scientific publications discussing "embedded cloud". However, most of the findings are not computer system studies. IEEE Xplore search engine does not give hits for the "embedded cloud" term; a search with separated terms over AND operation gives 329 hits from the relevant research fields.

Current "embedded", IoT, or sensor cloud proposals utilize IoT devices only as heterogeneous data providers and use existing cloud computing approaches for homogenizing and further refining the data for the end-user applications [3, 8, 14]. In such papers, the embedded cloud is seen as an assorted collection of homogenized data producers. However, we consider embedded cloud to have more requirements from the IoT devices. Therefore, we propose a definition for the embedded cloud.

Cloud computing provides virtualized computing services without logical or location relation to physical hardware [10]. This is impossible in the embedded world due to tight relation with the physical world: sensors measure specific quantities at the specific locations and at the specific time [2]. Thus, the embedded cloud has two paradigms: (1) the heterogeneous IoT devices provide homogenized services to the end user through the embedded cloud, such as measurement data and actuator controls. (2) The embedded cloud provides and shares services between the IoT devices, such as processing power, storage space, and those services that do not exist on a particular IoT device. Then, the IoT device itself can virtually extend its resource to the embedded cloud. If these two paradigms are accomplished, creating ubiquitous applications could be revolutionized, since the resource constraints of the smallest IoT devices would no longer restrict the development, not even at the local domain of the device itself. Figure 1 illustrates this definition of the embedded cloud.

The general requirements for an embedded cloud design are as follows.

(1) It should homogenize data accessing and processing of heterogeneous IoT technologies for the end user.

(2) It should extend resources of the IoT technologies.

(3) It should distribute processing that the IoT technologies can share resources and make the most of the available resources.

All three requirements are complicated due to the heterogeneous IoT devices with varying computing, communication, and energy resources. First, homogenization is difficult, since communication models, sampling rates and accuracies, data access notations, and so forth are different between technologies. Second, extending the resources is difficult, for example, if the device is only capable of sending and receiving few bytes occasionally. Third, distributing the processing and sharing resources require approaches where computation,

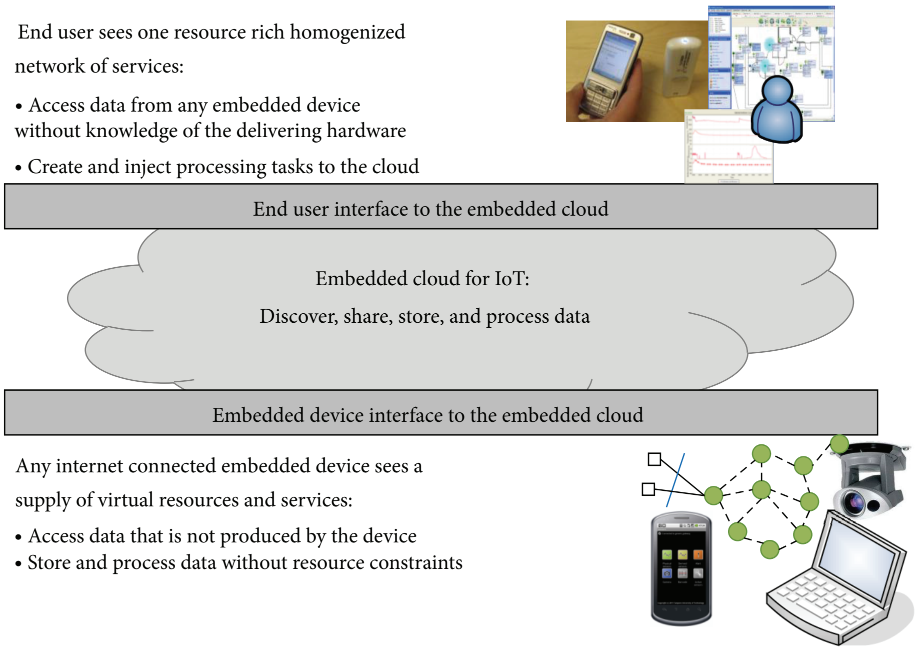

End user sees one resource rich homogenized network of services:
• Access data from any embedded device without knowledge of the delivering hardware
• Create and inject processing tasks to the cloud
End user interface to the embedded cloud
Embedded cloud for IoT:
Discover, share, store, and process data
Embedded device interface to the embedded cloud
Any internet connected embedded device sees a supply of virtual resources and services:
• Access data that is not produced by the device
• Store and process data without resource constraints

Figure 1: The concept of the embedded cloud.

communication, and energy resource differences are solved. The resulting design should be lightweight to work on the most resource-constrained IoT devices; yet, the design should be agile and versatile to cover the vast application space of IoT. Finally, through these requirements the embedded cloud should allow easier application development for the IoT technologies without the limitations of the resource constraints and without tailoring due to the technology heterogeneities.

3. Related Work

The critical problem in current heterogeneity solving proposals is the lack of harnessing the processing capabilities of the connected IoT technologies. They fold into two categories: first, an adaptation layer is used as a common gateway for heterogeneous technologies. In these solutions, such as OGC SWE [15] and GSN [16], the data produced and/or consumed by the sensor networks is adapted for the end-user application that runs on a server infrastructure [4, 17]. These solutions do not provide real device-to-device or technology-to-technology interoperation and cannot solve the basic use case: how to allow one device to utilize a measurement of a different device. They only remove the heterogeneity of the connected technologies for the end-user application. Further, the processing capabilities of the technologies are not harnessed. Second, a knowledge sharing points allow heterogeneous devices to interchange data, as in Smart-M3 [18]. These solutions do not share processing, extend resource, or harness the processing capabilities and they require tailored application development on each device.

Current sensor cloud or IoT cloud proposals use IoT devices as data producing sensors and data consuming actuators similar to heterogeneity adaptation proposals. IoT clouds use cloud computing for storing data, refining data, and providing refined services to end users; such proposals can be found, for example, in [14, 19–23]. These proposals use the cloud computing due to two factors. First, the cloud computing can dynamically scale to the ever increasing device, data, and end-user amounts, since IoT can potentially have many thousands of devices producing vast amount of data. Second, the cloud computing provides a cost-effective infrastructure to process the vast amount of data, while scaling to the random requests from the end users.

RoboEarth project proposes similar approaches for autonomous mobile robots. RoboEarth stores record of the physical world and action recipes to a database to be shared between Internet connected robots [24]. Our design is for wider interaction between heterogeneous IoT devices. Also, our design is able to cover RoboEarth functionality, achieve it on smaller devices, and extend it with interaction of other nonrobotic embedded devices. Authors of this paper were not able to find any directly comparable IoT design from the scientific world that would expand and share resource and distribute processing on the measuring and actuating IoT devices. However, the components of our design have comparable proposals.

Knowledge sharing proposals can be compared to DiMiWa. Smart-M3 [18] allows device-to-device data exchange through shared knowledge points, but it does not expand resources virtually or allow distributed processing. Gómez-Goiri and López-de-Ipiña propose a distributed middleware that shares data between heterogeneous IoT devices using RDF [25] and triples [26]. In addition to sharing, data can be queried from the tripe space. Gómez-Goiri and López-de-Ipiña proposal does not distribute processing, and the Java ME implementation is complex and heavy for resource-constrained devices. Both middleware

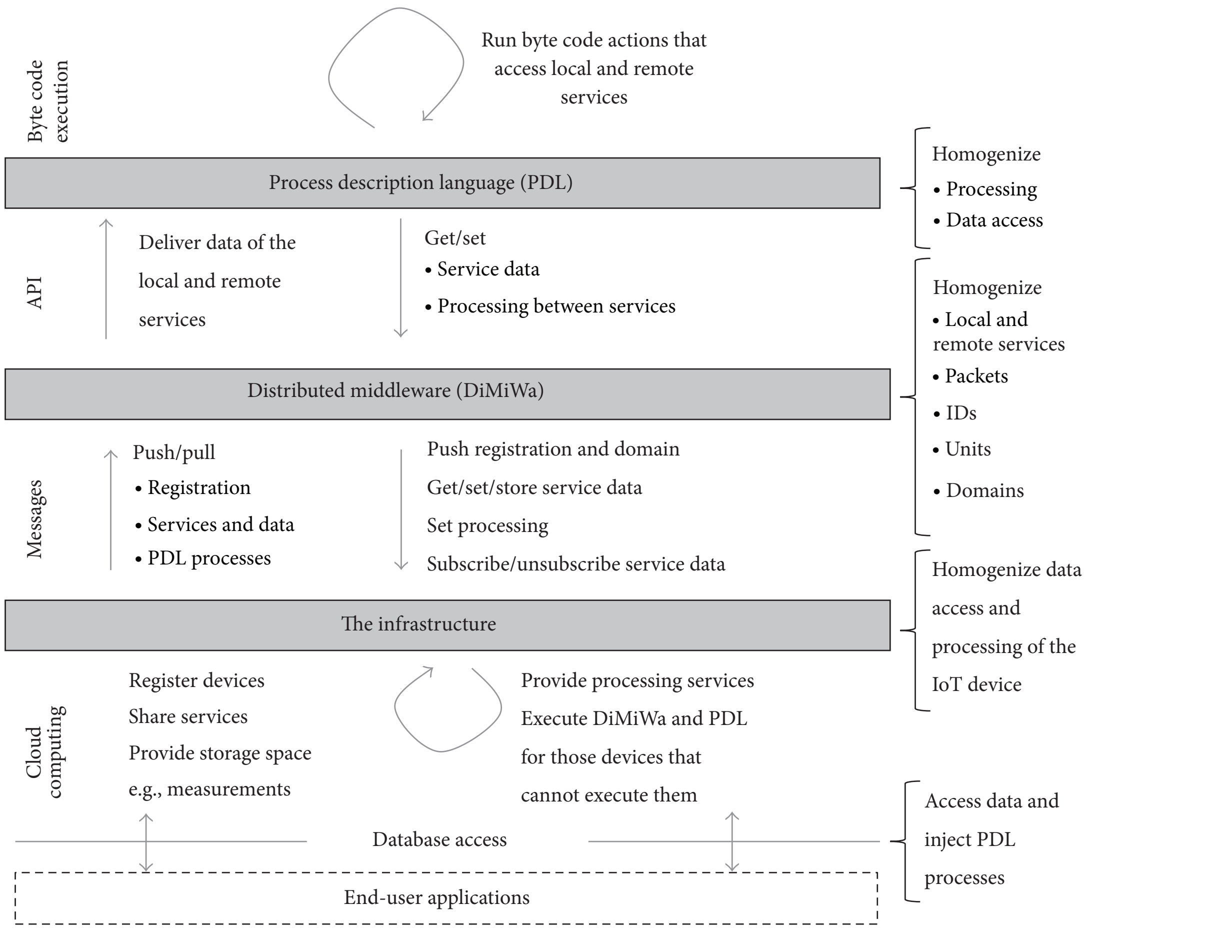

Byte code execution
Run byte code actions that access local and remote services
Process description language (PDL)
Homogenize
• Processing
• Data access
API
Deliver data of the local and remote services
Get/set
• Service data
• Processing between services
Homogenize
• Local and remote services
• Packets
• IDs
• Units
• Domains
Distributed middleware (DiMiWa)
Messages
Push/pull
• Registration
• Services and data
• PDL processes
Push registration and domain
Get/set/store service data
Set processing
Subscribe/unsubscribe service data
Homogenize data access and processing of the IoT device
The infrastructure
Cloud computing
Register devices
Share services
Provide storage space e.g., measurements
Provide processing services
Execute DiMiWa and PDL for those devices that cannot execute them
Database access
Access data and inject PDL processes
End-user applications

Figure 2: The embedded cloud design, its components, and the levels of homogenization.

proposals require tailored application implementation on the connecting IoT device.

Distributed WSN middleware can be compared to DiMiWa as well. SQL query-based middleware, such as TinyDB [27], provides a SQL-like interface to query and process measurements from a WSN; however, TinyDB cannot function between different technologies as DiMiWa does. TinyLIME [28] and Agilla [29] allow distributing data between mobile devices in a WSN through a tuple space. Again, these solutions cannot be distributed between different technologies. Any of the presented related solutions do not expand resources of the resource-constrained WSNs.

Virtual Machines (VMs) for WSNs and process description languages are the related work for the proposed PDL. Mate [30] is a well-known virtual machine for WSNs that allows distributing WSN applications to the resource-constrained WSN nodes without actual reprogramming. Mate itself does not support distributed processing and it functions close to the hardware. The proposed PDL allows process distribution and it utilizes abstracted services of DiMiWa. Process description languages are often XML, such as Web Service Definition Language (WSDL) [31], and intended for large scale computing. Thus, unlike PDL, they are not suitable for small resource-constrained IoT devices.

4. The Embedded Cloud Design

Our embedded cloud design consists of three layers as presented in Figure 2: the infrastructure, DiMiWa, and PDL. In this design, processes and services provide the homogenizing abstraction. A process is described as actions that use the services. Typical processing tasks (such as average calculation) are services as well. A process can be distributed onto the IoT devices where applicable. A domain is used to provide locality into the services. A device can only see those remote services that are in the same domain. Together these design components fill the requirements of the embedded cloud as shown in following sections.

4.1. The Infrastructure. The infrastructure design consists of four main parts: a database, communication handling, processing tasks, and end-user access. The communication between the infrastructure and DiMiWa is done via messages. The infrastructure should be driven on an Internet connected server platform, such as a cloud computing platform.

4.1.1. Messages. Table 1 describes messages that are sent by the infrastructure, and Table 2 describes messages that are sent by DiMiWa. Each message consists of message ID and payload,

Table 1: Messages from the infrastructure to the DiMiWa devices.

Message	Purpose
Push registration	Cloud informs device of successful registration, and pushes the domain to the device
Push domain	Cloud pushes a domain to the device
Push service	Cloud pushes a remote service to the device
Push service data	Cloud pushes data of the remote service to the device
Pull service data	Cloud pulls data of a service from the device
Flush services	Cloud instructs the device to flush current remote services (e.g., change in domain)
Push PDL process	Cloud pushes a new PDL process to the device
Flush PDL processes	Cloud instructs the device to flush current PDL processes (e.g., change in domain)

Table 2: Messages from the DiMiWa devices to the infrastructure.

Message	Purpose
Push registration	The device registers in to the embedded cloud
Push domain	The device informs of a location change to the embedded cloud
Update service	The device requests for new TTL for a service
Get service data	The device requests for new data for an existing service and data is delivered once
Set service data	The device sets new data for a service
Store service data	The device instructs cloud to store new data of an existing service. Data is stored once
Subscribe service	Subscribed service data is delivered continuously when new data is ready
Unsubscribe service	The device unsubscribes for data
Set processing service	The device sets a service for a processing service

and the payload typically constructed from service IDs, small amount of data, or a domain. The data formatting (units) is part of the implementation.

The messages allow the basic functionalities of service discovery, data delivery, and process injection. The service discovery happens during the registration. The IoT device informs about itself to the embedded cloud, which responses with the available services. The data delivery is performed with the service data pushing and pulling, getting and setting, and subscribing and unsubscribing. The distributed processing is done with pushing and pulling the processes by the infrastructure and creating processing service chains by the IoT device.

The communication parameters, such as node and network IDs, should be in the headers of the packet that encapsulates the message. An adapter may be required to translate the IDs, which is a part of the implementation of DiMiWa.

4.1.2. Database. Each registered device is stored in to Node DB that contains a node ID, a domain, a key to Network DB, and relations to Service DB and Process DB. Network DB contains connection information to each network: an IP-address, a port, and a packet wrapping format, if required. Service DB keeps track of services of each node and in each domain, and Process DB keeps track of PDL processes. The Blob DB stores the data delivered to the embedded cloud and the data generated by the processing tasks in the infrastructure. The data has a time stamp of its arrival.

4.1.3. Processing. The infrastructure has three main processing tasks. First, it must handle the communication between devices running DiMiWa. Second, it must execute DiMiWa and PDL for those technologies that cannot execute them on their own. Third, it must run processing services that are too heavy to run on typical IoT devices, for example, FFT, pattern recognition, and artificial intelligence.

The communication handler approves node registrations, delivers domains and services to the nodes, stores and requests data to the blob store, handles the data subscriptions, and delivers data to the nodes. If one node instructs *store* or *get* to a remote service and the cloud does not have recent enough data from that service, the infrastructure can request a new data from the service owner.

The in-cloud processing and resource sharing create vast possibilities of distributed intelligence similar to RoboEarth [24]. For example, one could create a small battery-operated mobile robot with a camera and run a face detection on that device. If the mobile robot would be connected to the embedded cloud, it could store every face in to the cloud. Further, the face detection processing could be moved to the cloud as well, if this would, for example, save energy. Even further, if the robot is replicated, all the robots could share the same face database, and when one robot adds a new face to the cloud, all the robots would be instantly able to detect that same face.

4.1.4. End-User Access. We selected not to concentrate on end-user application interface in this paper, since it is a large research problem on its own. The embedded cloud essentially provides an access to its databases for the end user. In addition, the end user can add PDL processes and processing tasks to the infrastructure. However, the implementation is not straightforward or well abstracted with a such basic interface. We reckon that additional application cloud infrastructure would be beneficial to improve data refinement and allow higher level processing. This infrastructure could potentially generate PDL processes automatically from a high level description of an application, as we presented in [4]. This application cloud on top of the embedded cloud is left as future work.

4.2. Distributed Middleware. DiMiWa works as a middleware in the embedded cloud design that allows different technologies to connect to the embedded cloud. It provides a homogenized service interface for the PDL processes. The implementation can be executed on device or on

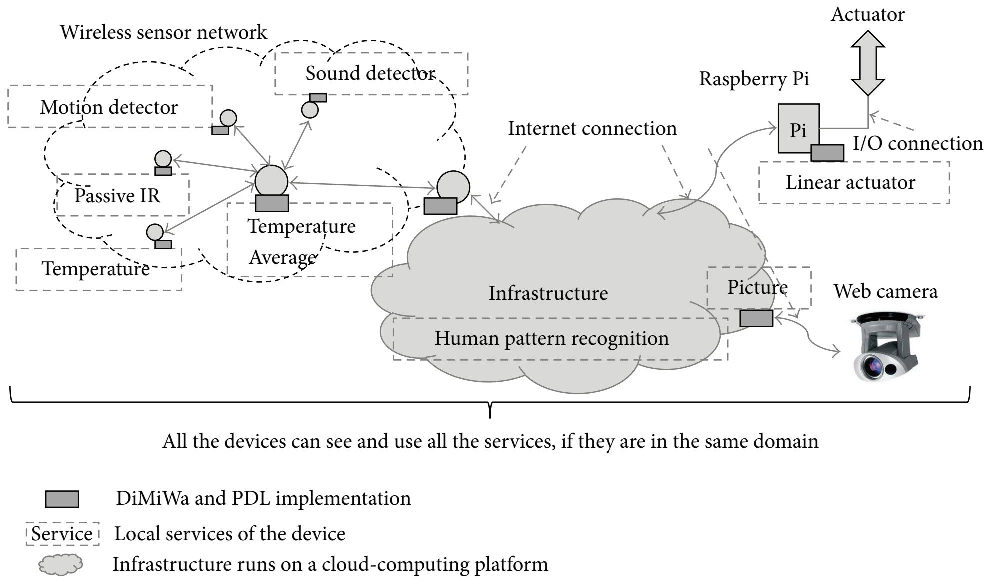

Wireless sensor network
Sound detector
Motion detector
Passive IR
Temperature
Temperature
Average
Internet connection
Actuator
Raspberry Pi
Pi
I/O connection
Linear actuator
Infrastructure
Picture
Web camera
Human pattern recognition
All the devices can see and use all the services, if they are in the same domain
DiMiWa and PDL implementation
Service
Local services of the device
Infrastructure runs on a cloud-computing platform

FIGURE 3: Possible execution locations for the DiMiWa and PDL implementations for a WSN, an Internet connected web camera and a linear actuator that has no execution capabilities.

the cloud, which ensures that no technology is discriminated. The implementation execution possibilities of DiMiWa and PDL are presented in Figure 3. If a device/technology cannot satisfy the presented requirements of DiMiWa, the device/technology can be connected to the cloud through an intermediate hardware, for example, with a Raspberry Pi as the actuator is connected in Figure 3. If the device/technology has an Internet connection capability, a virtual DiMiWa implementation can be run directly in the infrastructure, as the web camera is connected in Figure 3. Thus, DiMiWa is *distributed* middleware over several technologies.

To ensure wide portability of DiMiWa, the requirements for the executing IoT device have been minimized: the IoT device must be able to send and receive data packets from the Internet (e.g., through a tailored gateway) with a payload of at least 9 B (1 B packet type and at least 8 B of payload), it must provide a time stamping method, and it must be able to execute a DiMiWa implementation. As the only requirement is 9 B of payload and two-way communication, DiMiWa can be used on top of several IoT communication technologies, such as often referred ZigBee [32], 6LoPWAN [33], and CoAP [34].

DiMiWa consists of two interfaces: an application programming interface (API) for PDL and the message interface that is used to communicate with the infrastructure. The design is kept minimal to ensure wide portability. In addition to these two interfaces, a *service cache* is used to keep track of local and remote services. DiMiWa implementation can have technology specific features; for example, a DiMiWa implementation for a WSN could deliver data straight from node to node and bypass the infrastructure to reduce packet amount, or a WSN could select process runner according to the routing hierarchy and available in-network services.

4.2.1. An API to PDL. A PDL process can store a value of a service, trigger to a service, get a value of a service, set a value of a service, and instruct a service or data as an input to a processing service. The API is presented in Table 3. PDL uses these interface functions in its execution and the PDL design follows this same service paradigm. However, the IoT device can implement other applications on top of the DiMiWa API as well.

TABLE 3: DiMiWa API for PDL.

Operation	Parameters	Purpose
Store	Service	Stores value of the service for the calling device
Trigger	Service	Triggers to the service
Get	Service, buffer for data	Returns a value of the service
Set	Service, data in a buffer	Sets a value of the service
Get class	Service	Gets the class of the service
Process data	Service, data in a buffer	Sets data to be processed with the service
Process service	Service, service	Sets value of the service to be processed by another service

4.2.2. Service. A service contains an identifier, a domain, and a class. The service identifier and the domain together form an address space that is used to distinguish and access the services. The physical relation is abstracted; thus, the service user cannot know which physical device is actually implementing the service. However, the identifier and the domain restrict the physical location of the service implementer.

The service class contains a flag for local and remote services. The class itself provides information of the usable actions on the DiMiWa process. The classes are BLOB, SAMPLE, and EVENT. A BLOB class service produces amount

Table 4: PDL actions, parameters and purpose.

Action	Parameters	Purpose
STORE	SERVICE	Stores value of the SERVICE to the cloud
GET	SERVICE	Returns a value of the SERVICE. The value is stored to the internal ACCU
SET	SERVICE	Sets the ACCU to the SERVICE
SET_ACCU	DATA: a new internal value	Sets the immediate data to the ACCU
TRIGGER	SERVICE	Blocks process until the SERVICE returns a triggered condition
TIMER	DATA: wait time in seconds	Blocks process until the amount of seconds of the given data have passed
TIMEWINDOW	DATA: time window in seconds	Restarts the process, if the time window is not cleared before expiration
RESTART	—	Restarts the process
JUMP	DATA: a jump offset	Take a jump of the offset length
CONDITIONAL_JUMP	DATA: a jump offset	If the next ACTION will rise the TRUE flag, the process takes the jump. Offset is added after the conditional evaluation
PROCESS_SERVICE	process SERVICE, source SERVICE	Give the source SERVICE as an input to the process SERVICE
PROCESS_ACCU	process SERVICE	Give the ACCU as an input to the process SERVICE
OPERATION	SERVICE, OPERATION	Execute "SERVICE OPERATION ACCU" equation. Set TRUE flag if needed.
OPERATION_IMMEDIATE	SERVICE, OPERATION, DATA	Execute "SERVICE OPERATION DATA" equation. Set TRUE flag if needed.
INC_ACCU	—	Increments ACCU with 1.
DEC_ACCU	—	Decrements ACCU with 1.

of data that cannot be presented in one variable, for example, images, graphs, or sounds. The intention is that resource-constrained devices do not try to access a BLOB class services (it is not forbidden though). SAMPLE class services produce a measurement or take an adjustment parameter of the size of one variable on request or on intervals. EVENT class services produce measurements of the size of one variable after some events.

4.2.3. *Service Cache.*

Node mobility is evident in the embedded cloud, which causes services to appear and disappear dynamically: storing and discovering methods are needed. A service cache is used to store local and remote services and their data. DiMiWa registers local services in the infrastructure upon a registration. As a response, the infrastructure delivers available remote services according to the domain. All used services are stored in the service cache, and they all have a Time-To-Live (TTL) value that ensures cleaning of disappearing remote services eventually. The local services have an infinite TTL, and they stay permanently in the service cache.

The service cache holds the newest value for each SAMPLE and EVENT services. BLOB services are stored only into the infrastructure. The *get* operation returns the value for the SAMPLE and EVENT class services. The *trigger* returns a true boolean value, if there is a value entry with recent enough time stamp. The time threshold can be selected in the implementation taking technology specific packet delays into account. In a case of *store* operation for a local service, the value from the cache is sent to the infrastructure.

4.3. *Process Description Language.*

PDL was designed to be platform and programming language independent, to have a small memory footprint when implemented, and to not require real multitasking to ensure the suitability for the resource-constrained IoT devices. PDL itself provides a cooperative multitasking for its processes.

A PDL process is a series of known size actions that interact with DiMiWa services. Each action in the series is executed step by step in a state machine. PDL resembles an instruction set of a virtual machine, but the operands are local or remote services instead of typical register and memory accesses of a CPU. The known size of the actions and the following operands make parsing and implementing the executing state machine easy. Each process has an accumulator register ACCU for storing and manipulating values returned by the services.

Table 4 presents all the actions of PDL. The actions allow accessing the DiMiWa services, creating timed actions, manipulating the execution flow, manipulating the accumulator, and instructing the DiMiWa processing services. The small amount of the actions ensures that the implementation is lightweight.

PDL requires a call of a timing function once a second that is the timing granularity. If the target technology cannot provide this granularity, the PDL must be implemented in the infrastructure.

The execution flow can be manipulated with two different actions. First, a timeout can be created with TIMEWINDOW. If the following action does not proceed or produce a TRUE result within the time window, the process is restarted.

Second, a jump can be taken with three actions: jump always with JUMP, jump after CONDITIONAL_JUMP if the following action produces a TRUE, and jump in the beginning with RESTART. The conditionals TRUE and FALSE are set by the operation actions after the evaluation.

The accumulator can be manipulated with setting, adding, subtraction, multiply and division operations. The manipulation can be done between an immediate value or a DiMiWa service. The arithmetics are saturating; thus, accumulator overflow is not possible.

The end user can add new processes to the embedded cloud. The embedded cloud then resolves the best execution place for that process. The process could be even split into smaller PDL action series, if required. However, algorithm and design for splitting the PDLs are left as future work.

5. Evaluation with a Prototype

The evaluation studies the feasibility of the presented embedded cloud design. First, we evaluate the implementation feasibility of PDL and DiMiWa on resource-constrained IoT devices. Second, we discuss scalability issues. Third, we evaluate usability of the PDL processes using for example use cases. Finally, we present a comparison of features to existing related work.

5.1. Implementation Feasibility. We have implemented a portable implementation of DiMiWa with the C programming language. The portability requires two interfaces in addition to the API and messages of the embedded cloud design: a platform interface and a portable interface. The platform interface must be called by the IoT device running the DiMiWa implementation. The portable interface must be implemented by the IoT device.

The platform interface has three functions: handling the received packets, keeping up the connection to the embedded cloud, and resolving the domain, if the device physically moves.

The portable interface implements the following functions. An initialization function does all the device specific initializations, for example, adding the local services to the DiMiWa service cache. A wrapped packet allocation is provided, where the packet holds room for the IoT technology specific communication and the DiMiWa packet and the port fills the device specific headers and trailers. A function for sending a packet towards the Internet is provided. A function to get the time for DiMiWa cache is implemented. It should be noted that the time format can be anything, since the time is not delivered to the embedded cloud. Getting and setting functions are implemented for the values of the local services implemented by the device. Finally, memory allocation and freeing functions are implemented.

PDL implementation is standard C programming language code and can be directly compiled for any platform that has a C compiler. The implementation is 354 lines of C code including 15% of comment lines. One process data structure holds a container pointer, a process description pointer, a process description size, a location of the process

```
uint8_t clock(lc_t* lc, event_t* event){
  os_thread_begin(lc);
  while(1){
    os_alarm_set(1000); // 1ms granularity
    os_thread_wait_event(lc);
    pdle_clock_tick();
  }
  os_thread_end(lc);
}
uint8_t run(lc_t* lc, event_t* event){
  os_thread_begin(lc);
  while(1){
    os_alarm_set(100);
    os_thread_wait_event(lc);
    pdle_run( );
    dimiwa_run( );
  }
  os_thread_end(lc);
}
```

Listing 1: PDL and DiMiWa execution with threads of HybridKernel.

execution, status flags, a timer, a jump offset register, and an accumulator register. This yields 12 B and the size of two pointers of memory per each PDL process. One PDL action is 8 bits, jump offset is 8 bits, immediate data is 32 bits, and timers are 16 bits.

PDL provides four interface functions: pdle_clock_tick() must be called once a second to create the internal timer, pdle_add_process() adds a new process to the execution, pdle_remove_process() removes a running process from the execution, and pdle_run() executes the running processes.

In the DiMiWa implementation, one service is 32 bits, one data entry is 32 bits integer, the class is 8 bits, and the domain is 8 bits. One DiMiWa cache entry requires memory for 10 B of data (a service, a class, a TTL, a value, and a time stamp) and two pointers.

Listing 1 presents how PDL and DiMiWa are executed using threads in our HybridKernel [35]. These threads are built on protothreads proposed by Dunkels et al. [36], which are used in Contiki [37]. Thus, the same execution model should work for Contiki as well. Both systems software are designed for small 8-bit microcontrollers that are often used on resource-constrained WSN devices.

Our prototype implementation was tested using two PIC18F8722 8-bit microcontrollers [38] equipped TUTWSN WSN devices [39], two Raspberry Pi devices [40], and a laptop running the infrastructure as shown in Figure 4. The WSN devices implemented a Passive Infra-Red (PIR) motion detector and temperature measurement services, one Raspberry Pi implemented a camera service with USB WEB camera, and the other one implemented a sound producing on/off actuator service. The packets were delivered using UDP between the Internet connected devices.

On the TUTWSN device, the PDL implementation takes 1900 B, and DiMiWa implementation takes 3222 B in program memory. This totals to 5122 B, which is 4% of the available

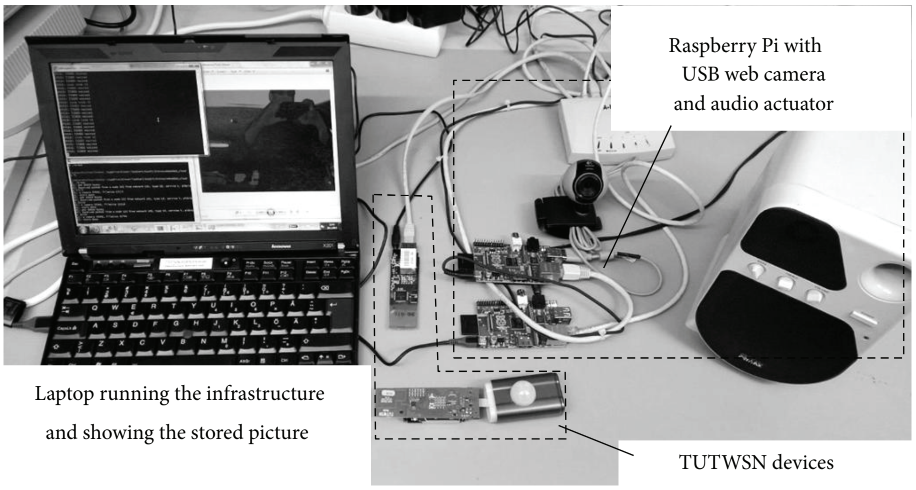

Raspberry Pi with USB web camera and audio actuator
Laptop running the infrastructure and showing the stored picture
TUTWSN devices

Figure 4: The prototype testing setup.

128 KB program memory. These values were gathered with Microchip MCC18 compiler using optimizations. The values do not contain the library functions (such as a linked list), drivers, protocol stack, or operating system. From the implementation results it can be concluded that PDL and DiMiWa can be implemented for a small 8-bit IoT device. The Raspberry Pi implementation executable is a 23.5 KB ELF file.

Executing a process on PDL requires varying time on each step and iteration. Table 5 presents the worst case execution times averaged for the PIC18F8722 implementation running at 4 MHz (1 instruction per μs). The execution times were gathered by probing MCU pins with an oscilloscope, running a test PDL process, and toggling pins according to actions. Jumps and operations manipulating ACCU require varying time due to the 8-bit MCU and 32-bit operations that are implemented with software by the compiler. The figures are considered reasonable, and the presented implementation can run over 100 PDL processes on the PIC18F8722 as shown in the scalability section. The execution times do not contain protocol stack operations (e.g., sending a packet).

Table 5: The worst case execution times of DPL actions on PIC18F8722 MCU.

Action	Execution time (μs)
TIMER and TIMEWINDOW when set	350
STORE, SET, PROCESS_SERVICE, PROCESS_ACCU	<800
TRIGGER, GET	190
idling (waiting for timer or RESTART)	180
rest of the actions (actions manipulating ACCU and jumps)	<400

5.2. Scalability. Scalability should be considered for small resource-constrained WSNs and IoT technologies, since their memory, energy, and communication resources are often limited. Thus, resource-constrained WSNs are the possible bottlenecks of the presented design. The main scalability issue is the amount of services and processes that one device can potentially handle. Eventually, data memory and execution time will run short.

With the presented TUTWSN and PIC18F8722 implementation, DiMiWa and PDL have around 1968 B of data memory in their use. As described in Section 5.1, one service entry in DiMiWa cache requires 10 B of data memory and two pointers. Since pointers are 2 B on PIC18F8722, this totals to 14 B per cache entry and the upper limit for subscribed services of one device is 1968 B/14 B = 140. The PDL processes can be stored to the program memory, but each PDL process requires 16 B of data memory and the upper limit for processes is 123. Since both processes and services compete from the same data memory resource, a compromise is needed, and both the process and the service amounts need to be carefully controlled.

The execution time depends on the PDL process amount and structure. The presented implementation runs PDL every 100 ms (Listing 1), and on the worst case one action takes 800 μs. As a result, the deadline of 100 ms will be breached with 125 PDL processes running a lengthy action *at the same step*. Considering the data memory constraints, it is seen that the execution time will not restrict scalability on PIC18F8722-based platforms. However, the execution time is energy consuming, and therefore the PDL process amount should be carefully controlled.

The presented figures suggest that the implementation is well usable with PIC18F8722 and a TUTWSN network of 100 devices could execute even 12300 different PDL processes, which should be considered enough. Resource-constrained WSNs typically have limited amount of duties, since the connections (sensors and actuators) to the physical world are limited.

In addition to service and PDL process amounts, the message exchange affects scalability. If the device subscribes a lot of remote services, the infrastructure might push too much data to the device. This could cause two drawbacks: first, the network gets congested due to the amount of delivered

```
uint8_t pdl_process[ ] = {
    PDL_ACTION_TIMER, 0 × 01, 0 × 2C,
    PDL_ACTION_STORE, DIMIWA_SERVICE_TEMPERATURE,
    PDL_ACTION_RESTART };
```

Listing 2: A 9-byte PDL process for measuring temperature every 5 minutes.

```
uint8_t pdl_process[ ] = {
    PDL_ACTION_TRIGGER, DIMIWA_SERVICE_PIR,
    PDL_ACTION_TIMEWINDOW, 0 × 00, 0 × 3C,
    PDL_ACTION_TRIGGER, DIMIWA_SERVICE_PIEZO,
    PDL_ACTION_TIMEWINDOW, 0 × 00, 0 × 3C,
    PDL_ACTION_TRIGGER, DIMIWA_SERVICE_SOUND,
    PDL_ACTION_STORE, DIMIWA_SERVICE_CAMERA,
    PDL_ACTION_PROCESS_SERVICE, DIMIWA_SERVICE_HUMAN_PATTERN_RECOGNITION,
    DIMIWA_SERVICE_CAMERA,
    PDL_ACTION_CONDITIONAL_JUMP, 0 × 13,
    PDL_ACTION_OPERATION_IMMEDIATE, DIMIWA_SERVICE_HUMAN_PATTERN_RECOGNITION,
    PDL_OPERATION_EQUAL  , 0 × 00, 0 × 00, 0 × 00, 0 × 00,
    PDL_ACTION_SET_ACCU, 0 × 00,0 × 00,0 × 00,0 × 01,
    PDL_ACTION_SET, DIMIWA_SERVICE_GATE_ACTUATOR,
    PDL_ACTION_TIMER, 0 × 01, 0 × 2C,
    PDL_ACTION_DEC_ACCU,
    PDL_ACTION_SET, DIMIWA_SERVICE_GATE_ACTUATOR,
    PDL_ACTION_RESTART };
```

Listing 3: A 66-byte PDL process that detects human from a camera picture after motion detection sensor alarms and controls a gate similar to the constellation in Figure 3.

messages. Second, the increase in data traffic might increase energy consumption. These figures depend on the underlying transport technology and routing topology, which make estimating them very difficult as the embedded cloud is designed to connect any technology. For example, on TUTWSN, the increase in data traffic is not vital, if the messages are delivered through the so-called reserved slots [39]. Communication must be done on these slots even if there is no application data to be send. However, it must be emphasized that the presented design allows the implementation of DiMiWa and/or the execution of the PDL process resides on the infrastructure as shown in Figure 3 with the web camera. This would remove the overheads from the technology. In the current design, this issue can be avoided with careful use of the PDL processes.

The scalability of the infrastructure is assumed to be infinite in this paper. In practice, there is an upper limit for amount of connected devices, exchanged data, and services, but we assume that exhausting the infrastructure is a nonrelevant problem, if the design is deployed in large scale over distributed data centers or cloud platforms; for example, Facebook is currently able to serve over one billion users [41].

5.3. Example Use Cases. With the following three use cases, we show the versatility, usability, and feasibility of the PDL processes. Listing 2 presents a basic temperature measurement process with 5-minute intervals. This is a typical task required from a WSN node. Listing 3 presents a process that detects humans from a picture after three distinctive motion detectors have been triggered within a time frame of one minute. If a human is detected, a gate is opened for 5 minutes. This process could be used to separate humans from caged wildlife. Implementing this process on a resource-constrained WSN would be difficult due to the large data of the picture and processing power required by the image processing. Listing 4 presents a simple P controlled temperature controlling that utilizes averaged temperature, which would improve the result compared to a single point of measure. This is a typical WSN middleware and in-network processing use case for example, in a building automation.

Listing 2 process is in size 9 B. Respectively, Listing 3 is 66 B in size and Listing 4 is 62 B in size. Direct comparison to a tailored application or a Maté implementation is difficult due to the service approach of DiMiWa, but it is easy to estimate that programming the same behavior with CPU like byte code would yield a larger footprint. Relatively small size of the processes allows a process injection to a WSN over the WSN protocol stack, without a need for a program image or firmware distribution support [42]. The use cases were tested with the prototype implementation by simulating the physically missing hardware.

On Listing 3, the triggering could be executed on the node that is the first node to route all three triggering messages,

```
(1) uint8_t pdl_process[ ] = {
(2)     PDL_ACTION_TIMER, 0 × 01, 0 × 2C,
(3)     PDL_ACTION_PROCESS_SERVICE, DIMIWA_SERVICE_AVERAGE, DIMIWA_SERVICE_TEMPERATURE,
(4)     PDL_ACTION_TIMER, 0 × 01, 0 × 2C,
(5)     PDL_ACTION_CONDITIONAL_JUMP, 0 × 0B,
(6)     PDL_ACTION_OPERATION_IMMEDIATE, DIMIWA_SERVICE_AVERAGE, PDL_OPERATION_LT,
        0 × 00, 0 × 00, 0 × 08, 0 × 98,
(7)     PDL_ACTION_GET, DIMIWA_SERVICE_HEATING,
(8)     PDL_ACTION_DEC_ACCU,
(9)     PDL_ACTION_SET, DIMIWA_SERVICE_HEATING,
(10)    PDL_ACTION_CONDITIONAL_JUMP, 0 × 0B,
(11)    PDL_ACTION_OPERATION_IMMEDIATE, DIMIWA_SERVICE_AVERAGE, PDL_OPERATION_GT,
        0 × 00, 0 × 00, 0 × 08, 0 × 34,
(12)    PDL_ACTION_GET, DIMIWA_SERVICE_HEATING,
(13)    PDL_ACTION_INC_ACCU,
(14)    PDL_ACTION_SET, DIMIWA_SERVICE_HEATING,
(15)    PDL_ACTION_RESTART };
```

Listing 4: A 62 bytes PDL process with *P* controlled temperature controlling.

Table 6: Comparison of features.

Technology:	Presented proposal	Smart-M3	Mate	TinyDB	OGC SWE	RoboEarth
Technology interoperation	X	X				X
Heterogeneity homogenization	X	X		(partly)	X	X
Application implementation	X		X	(partly)		X
Application dissemination	X		X	(partly)		X
Distributed processing	X			X		X
Resource expanding	X					X
Usable directly on 8-bit WSNs	X	X	X	X		
Last measurement data access	X	X	X	X	X	
Archived measurement data access	X	X		X	X	

for example, the first routing node in Figure 3. When the three motion detection sensors alarm (PIR, motion sensor, and sound sensor) and send their messages to the routing node, the node could drop the packets and only send the store command for the camera. Typically, this would reduce communication, energy consumption, and/or congestion [4–7]. The average calculation on Listing 4 process could be executed with in-network processing of a WSN with the same benefits.

5.4. Comparison. Comparing the memory and execution overheads of the implementation is difficult, since it would require implementing all the related work on the same platform and there are no existing comparable overhead figures. Also, the novelty of the presented design is in its features that make it versatile. Therefore, we concentrate to compare features of related work to present our design versatility in Table 6. Although RoboEarth is not intended for IoT use, it has been included since it shares similar approach. TinyDB supports application implementation, dissemination, and distributed processing within one technology; thus, it is only considered to support those partly.

6. Open Questions and Discussion

The presented design is a starting point for the embedded cloud designs. There are open questions; for example, security, privacy, ontologies, or data semantics are main requirements for any IoT technology. Security and privacy are not discussed, since the presented design and implementation rely on the built-in security methods of the used technologies. Handling the encryption keys and so forth is an open problem that should be solved. Ontology and semantics rule the format and units of the data. One ontology can be selected in the implementation of the presented design. The ontology should provide identifiers for each device, and it should define units, ranges, accuracy, and so on for each measurement in the embedded cloud. The units and accuracies were selected according to the available hardware on the prototype implementation.

Although connected devices register to the embedded cloud through DiMiWa and do the service discovery with help of the infrastructure, there are management issues that were not discussed in this paper. For example, how the domains are set. In the prototype implementation we hardly coded the devices and their domains, but an automated method is needed to make this design more complete. This would be easy to solve, if all the devices could produce a WGS84 or similar coordinates for their location. On the other hand, often the domain is a description of the place, such as kitchen, a social security number, or a room number. Creating a completely automated system is a very difficult research problem.

The presented design could be improved for even better scalability with two extra components that keep track on subscribed services and running PDL processes on each device. These components could be called *DiMiWa service and PDL process brokers*. The DiMiWa service broker could monitor the amount of services subscribed on the device, and when the threshold is achieved, it could move some of the processing to the infrastructure to reduce amount of subscribed remote services. The PDL process broker could ensure that the device does not exhaust under its load. The PDL process broker could move PDL processes to the infrastructure, if the device runs out of memory, execution time, or energy. These two brokers should cooperate to ensure the best utilization of the technologies. Therefore, they both require understanding of the executing technology, but currently there are no such technology models available that could be used as an input data for these brokers. The brokers and the technology model are the important part of our future work.

7. Conclusions and Future Work

In this paper, an embedded cloud was defined as a design that homogenizes distributed processing, resource extending, resource sharing, and data accessing of heterogeneous IoT devices. A novel embedded cloud design was presented that consists of a distributable process description language, a distributable middleware, and an infrastructure. Together these components form an embedded cloud that expands resources, distributes processing, and hides heterogeneity even for the most resource-constrained IoT technologies. The prototype implementation was functional even on an 8-bit microcontroller-based WSN device.

As future work, we will study modeling IoT devices to describe their performance and capabilities and to try to create an algorithm that utilizes these performance descriptions to distribute the PDL processes in a more intelligent way through the DiMiWa service and PDL process brokers.

References

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: a survey," *Computer Networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] L. Tan and N. Wang, "Future internet: the internet of things," in *Proceedings of the 3rd International Conference on Advanced Computer Theory and Engineering (ICACTE '10)*, vol. 5, pp. V5376–V5380, August 2010.

[3] P. Parwekar, "From internet of things towards cloud of things," in *Proceedings of the 2nd International Conference on Computer and Communication Technology (ICCCT '11)*, pp. 329–333, September 2011.

[4] T. Laukkarinen, J. Suhonen, and M. Hannikainen, "A survey of wireless sensor network abstraction for application development," *International Journal of Distributed Sensor Networks*, vol. 2012, Article ID 740268, 12 pages, 2012.

[5] Q. Luo, H. Wu, W. Xue, and B. He, "Benchmarking in-network sensor query processing," Technical Report, The Hong Kong University of Science and Technology, Hong Kong, 2005.

[6] G. L. Prakash, M. Thejaswini, S. H. Manjula, K. R. Venugopal, and L. M. Patnaik, "Energy efficient in-network data processing in sensor networks," *World Academy of Science, Engineering and Technology*, vol. 24, pp. 12–25, 2008.

[7] C. Intanagonwiwat, D. Estrin, R. Govindan, and J. Heidemann, "Impact of network density on data aggregation in wireless sensor networks," in *Proceedings of the 22nd International Conference on Distributed Systems Workshops*, pp. 457–458, July 2002.

[8] A. Alamri, W. S. Ansari, M. M. Hassan, M. S. Hossain, A. Alelaiwi, and M. A. Hossain, "A survey on sensor-cloud: architecture, applications, and approaches," *International Journal of Distributed Sensor Networks*, vol. 2013, Article ID 917923, 18 pages, 2013.

[9] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Proceedings of the Grid Computing Environments Workshop (GCE '08)*, pp. 1–10, November 2008.

[10] M. Armbrust, A. Fox, R. Griffith et al., "A view of cloud computing," *Communications of the ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[11] Embedded Hardware, "40 posts categorized embedded cloud," 2013, http://blog.vdcresearch.com/embedded_hw/embedded-cloud/.

[12] Real-Time Innovations Inc and R. Joshi, "The embedded cloud: it at the edge," 2013, http://www.embedded.com/design/prototyping-and-development/4219526/The-embedded-cloud-IT-at-the-edge.

[13] H. Davis, "Embedded cloud computing," 2013, http://embedded.communities.intel.com/community/en/software/blog/2011/04/11/embedded-cloud-computing.

[14] G. C. Fox, S. Kamburugamuve, and R. D. Hartman, "Architecture and measured characteristics of a cloud based internet of things," in *Proceedings of the International Conference on Collaboration Technologies and Systems (CTS '12)*, pp. 6–12, May 2012.

[15] M. Botts, G. Percivall, C. Reed, and J. Davidson, "Ogc sensor web enablement: overview and high level architecture," in *GeoSensor Networks*, S. Nittel, A. Labrinidis, and A. Stefanidis, Eds., vol. 4540 of *Lecture Notes in Computer Science*, pp. 175–190, Springer, Berlin, Germany, 2008.

[16] K. Aberer, M. Hauswirth, and A. Salehi, "Infrastructure for data processing in large-scale interconnected sensor networks," in *Proceedings of the 8th International Conference on Mobile Data Management (MDM '07)*, pp. 198–205, May 2007.

[17] N. Mohamed and J. Al-Jaroodi, "Service-oriented middleware approaches for wireless sensor networks," in *Proceedings of the 44th Hawaii International Conference on System Sciences (HICSS '10)*, pp. 1–9, January 2011.

[18] J. Honkola, H. Laine, R. Brown, and O. Tyrkkö, "Smart-M3 information sharing platform," in *Proceedings of the 15th IEEE Symposium on Computers and Communications (ISCC '10)*, pp. 1041–1046, June 2010.

[19] M. M. Hassan, B. Song, and E. Huh, "A framework of sensor-cloud integration opportunities and challenges," in *Proceedings*

of the 3rd International Conference on Ubiquitous Information Management and Communication (ICUIMC '09), pp. 618–626, ACM, New York, NY, USA, January 2009.

[20] M. Yuriyama and T. Kushida, "Sensor-cloud infrastructure physical sensor management with virtualized sensors on cloud computing," in *Proceedings of the 13th International Conference on Network-Based Information Systems (NBiS '10)*, pp. 1–8, September 2010.

[21] M. Yuriyama, T. Kushida, and M. Itakura, "A new model of accelerating service innovation with Sensor-Cloud Infrastructure," in *Proceedings of the Annual SRII Global Conference (SRII '11)*, pp. 308–314, April 2011.

[22] M. S. Aslam, S. Rea, and D. Pesch, "Service provisioning for the wsn cloud," in *Proceedings of the IEEE 5th International Conference on Cloud Computing (CLOUD '12)*, pp. 962–969, June 2012.

[23] S. Alam, M. M. R. Chowdhury, and J. Noll, "SenaaS: an event-driven sensor virtualization approach for internet of things cloud," in *Proceedings of the 1st IEEE International Conference on Networked Embedded Systems for Enterprise Applications (NESEA '10)*, pp. 1–6, November 2010.

[24] M. Waibel, M. Beetz, J. Civera et al., "Robo earth," *IEEE Robotics and Automation Magazine*, vol. 18, no. 2, pp. 69–82, 2011.

[25] "Resource description framework (rdf)," 2013, http://www.w3.org/RDF/.

[26] A. G. Gómez-Goiri and D. López-De-Ipiña, "A triple space-based semantic distributed middleware for internet of things," in *Current Trends in Web Engineering*, F. Daniel and F. M. Facca, Eds., vol. 6385 of *Lecture Notes in Computer Science*, pp. 447–458, Springer, Berlin, Germany, 2010.

[27] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "TinyDB: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems*, vol. 30, no. 1, pp. 122–173, 2005.

[28] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. L. Murphy, and G. P. Picco, "TinyLIME: bridging mobile and sensor networks through middleware," in *Proceedings of the 3rd IEEE International Conference on Pervasive Computing and Communications (PerCom '05)*, pp. 61–74, March 2005.

[29] C.-L. Fok, G.-C. Roman, and C. Lu, "Mobile agent middleware for sensor networks: an application case study," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks (IPSN '05)*, pp. 382–387, April 2005.

[30] P. Levis and D. Culler, "Mate: a tiny virtual machine for sensor networks," *ACM SIGPLAN Notices*, vol. 37, no. 10, pp. 85–95, 2002.

[31] W3C, "Web services description language (wsdl) version 2.0 part 1: core language," 2013, http://www.w3.org/TR/wsdl20/.

[32] ZigBee Alliance, "Zigbee alliance homepage," 2013, http://www.zigbee.org/.

[33] IETF, "Ietf ipv6 over low power wpan (6lopwan) working group," 2013, http://datatracker.ietf.org/wg/6lowpan/charter/.

[34] IETF, "Ietf constrained application protocol (coap) working group," 2013, https://datatracker.ietf.org/doc/draft-ietfcore-coap/.

[35] T. Laukkarinen, V. A. Kaseva, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "Hybridkernel: preemptive kernel with event-driven extension for resource constrained wireless sensor networks," in *Proceedings of the IEEE Workshop on Signal Processing Systems (SiPS '09)*, pp. 161–166, October 2009.

[36] 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)*, pp. 29–42, Boulder, Colo, USA, November 2006.

[37] A. Dunkels, B. Grönvall, and T. Voigt, "Contiki-a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks (LCN '04)*, pp. 455–462, Tampa, Fla, USA, November 2004.

[38] Microchip, "Pic18f8722 family data sheet," 2013, http://ww1.microchip.com/downloads/en/DeviceDoc/39646c.pdf.

[39] M. Kohvakka, *Medium Access Control and Hardware Prototype Designs for Low-Energy Wireless Sensor Networks*, Tampere University of Technology, Publication 808, Tampere, Finland, 2009.

[40] Raspberry Pi, "Raspberry pi," 2013, http://www.raspberrypi.org/.

[41] Yahoo News, "Number of active users at facebook over the years," 2013, http://news.yahoo.com/number-activeusers-facebook-over-230449748.html.

[42] T. Laukkarinen, L. Määttä, J. Suhonen, T. D. Hämäläinen, and M. Hännikäinen, "Design and implementation of a firmware update protocol for resource constrained wireless sensor networks," *International Journal of Embedded and Real-Time Communication Systems (IJERTCS)*, vol. 2, no. 3, pp. 50–68, 2011.

Tampereen teknillinen yliopisto
PL 527
33101 Tampere

Tampere University of Technology
P.O.B. 527
FI-33101 Tampere, Finland

ISBN 978-952-15-3542-0
ISSN 1459-2045