UNIVERSITY OF LATVIA
FACULTY OF COMPUTING

Girts Strazdins

Wireless Sensor Network Software Design Rules

DOCTORAL THESIS FOR THE ACADEMIC DEGREE OF
DR.SC.COMP

FIELD:	COMPUTER SCIENCE
SUB-FIELD:	SYSTEMS OF DATA PROCESSING AND COMPUTER NETWORKS
ADVISOR:	DR.SC.COMP. LEO SELAVO

RIGA, 2014

LATVIJAS
UNIVERSITATE
ANNO 1919

IEGULDĪJUMS TAVĀ NĀKOTNĒ

This work has been supported by the European Social Fund within the project "Support for Doctoral Studies at University of Latvia".

Abstract

In the last decade wireless sensor networks (WSNs) have evolved as a promising approach for smart investigation of our planet, providing solutions for environment and wild animal monitoring, security system development, human health telemonitoring and control, industrial manufacturing and other domains.

Lack of unified standards and methodologies leads to limited sensor network solution interoperability and portability. Significant number of WSN operating systems, virtual machines, query languages and other software tools already do exist. Also a significant number of communication protocols have been invented. However, sensor network designers and programmers still face serious problems related to new platform and application development.

The goal of this work is to propose wireless sensor network software development design rules that serve as a unified methodology for operating system and application development. The design rules are based on 40 existing WSN deployment extensive analysis and common trend inference. The proposed rules are evaluated in different aspects. Improvements for existing WSN deployments and operating systems are identified, design and implementation of an object-oriented WSN operating system according to proposed rules is described. In addition, a WSN application use-case is evaluated and improvements are suggested according to design rules. The evaluation shows the proposed design rules as an important tool for WSN software development at different stages, from planning to testing and change request analysis.

Keywords: wireless sensor networks, methodology, design rules, operating systems, deployment survey, case study

To my grandfather Modris and son Martins.

Acknowledgements

I would like to thank my advisor Leo Selavo, who has introduced the beauty of research work to me and acted as a successful role model. My deepest appreciation to my wife for support during the long run. Thanks to my playful and active kids for not allowing my head to overheat. I'm grateful to my parents and grandparents for cultivating importance of education since my childhood, especially my mother, who motivated me to always strive for more. Thanks to my sister for always being there in critical moments; persisting in goal reaching; and showing sincere humanity. I'm thankful to my father for creating environment where my passion to information technologies could thrive; while enough severity was maintained to motivate me explore beyond computer games. Thanks to director of Institute of Electronics and Computer Science (EDI), Modris Greitans, for opportunity to work in research full time and meet many talented young researchers, especially Atis Elsts, Artis Mednis and Reinholds Zviedris, who supplied me invaluable technical and scientific advise, and with whom I had the honor to have inspiring informal discussions. Thank you to Sashidharan Komandur and Hans Petter Hildre for patience and support during the final stretch that took much longer than initially planned. I appreciate the patience and professional comments of Guntis Arnicans, Guntis Barzdins, Valerijs Zagurskis, Michael Huhns, Yushan Pan, Cong Liu and other reviewers who made this thesis better in multiple iterations. Warm thanks to secretaries Ella Arsa, Anita Ermusa and Ruta Ikauniece at the University of Latvia, who helped to solve many stressful situations, and always were armed with eternal smile and positive attitude. I express my gratitude to my elementary school teachers Sigita Blumfelde and Gunta Dance for awakening the interest to math in me; and Uldis Jansons for teaching me the fascinating first steps in BASIC programming environment. I'm thankful to Ilze Zarinova and Iveta Mezatuca for bringing me back on track every time I experienced serious health problems on the way. And finally, thanks to the reviewer who has improved the thesis significantly by not allowing to accept the first version. It was a long journey, full of reasoning, discoveries and surprises.

Contents

List of Figures 5

List of Tables 7

Glossary 10

1 Introduction 13

1.1 Project experience . . . 14
1.2 Scope and motivation . . . 15
1.3 Contribution of the thesis . . . 17
1.4 Related work . . . 18
1.5 Summary and thesis outline . . . 19

2 Sensor network software abstractions 21

2.1 Operating systems . . . 23
2.1.1 MansOS . . . 24
2.1.2 TinyOS . . . 27
2.1.3 Contiki . . . 28
2.1.4 LiteOS . . . 29
2.1.5 Mantis . . . 29
2.1.6 Arduino . . . 29
2.2 Middleware . . . 30
2.3 Summary . . . 31

3 Deployment survey 32

3.1 Methodology . . . 32
3.2 Survey results . . . 32
3.2.1 Deployment state and attributes . . . 37
3.2.2 Sensing . . . 38
3.2.3 Lifetime and energy . . . 39

3.2.4 Sensor motes . . . 41
3.2.5 Sensor mote: microcontroller . . . 41
3.2.6 Sensor mote: external memory . . . 42
3.2.7 Communication . . . 42
3.2.8 Network . . . 43
3.2.9 Networking stack . . . 44
3.2.10 Operating system and middleware . . . 45
3.2.11 Software level tasks . . . 45
3.2.12 Task scheduling . . . 47
3.2.13 Time synchronization . . . 47
3.2.14 Localization . . . 48
3.2.15 Real-time data access . . . 48
3.2.16 Discussion of future trends . . . 49
3.2.17 Summary . . . 51

4 Sensor network software design rules 52

4.1 Problem identification . . . 52
4.1.1 Portability and usability . . . 52
4.1.2 Wireless communication . . . 53
4.1.3 Services and efficiency . . . 53
4.2 Design rule definition . . . 54
4.2.1 Communication . . . 56
4.2.2 Portability . . . 57
4.2.3 Task scheduling . . . 58
4.2.4 Services . . . 59
4.2.5 User support . . . 59
4.3 Addressing problems by rules . . . 60
4.3.1 Portability and usability . . . 60
4.3.2 Wireless communication . . . 62
4.3.3 Services and efficiency . . . 62
4.4 Summary . . . 63

5 Design rule impact on existing systems 65

5.1 Impact on deployments . . . 65
5.2 Impact on operating systems . . . 67
5.2.1 TinyOS . . . 67
5.2.2 Contiki . . . 69

5.2.3 LiteOS 70
5.2.4 Mantis 70
5.2.5 MansOS 71
5.2.6 Arduino 72
5.2.7 Summary 73
5.3 Use case study: wearable sensor network 73
5.3.1 Research problem 74
5.3.2 Approach 74
5.3.3 Sensor network aspects 75
5.3.4 System prototype 76
5.3.4.1 Hardware components 76
5.3.4.2 Software components 78
5.3.5 Prototype conformance to design rules 78
5.3.5.1 Communication 78
5.3.5.2 Portability 79
5.3.5.3 Task scheduling 80
5.3.5.4 Services 80
5.3.5.5 User support 80
5.3.6 Improvements by matching design rules 81
5.3.6.1 Network lifetime extension 81
5.3.6.2 Multi-hop communication 83
5.3.6.3 Multitasking support 83
5.3.7 Use case summary 84
5.4 Summary 84

6 New operating system design according to rules 85
6.1 OOMOS' advantages over MansOS 85
6.2 Object-oriented programming for WSNs 86
6.3 OOMOS implementation 87
6.3.1 Portability 87
6.3.2 Scheduling 90
6.3.3 Services and API 92
6.3.4 Summary 93
6.4 OOMOS evaluation 93
6.4.1 RAM and flash memory usage 94
6.4.2 Performance 96
6.4.2.1 Sensor sampling performance 97

6.4.2.2 Wireless data transmission performance 98
6.4.2.3 Wireless data reception performance 100
6.4.3 Optimizations 101
6.4.4 Portability 104
6.4.5 Object-orientation overhead 111
6.5 Future work according to design rules 112
6.5.1 Networking protocol stack 113
6.5.2 Services and scheduling 115
6.6 Summary 115

7 Conclusion **116**

References **119**

Appendices **131**

A WSN deployments **132**
A.1 Application taxonomy 132
A.2 Deployment survey detailed results 134

B Hardware platform survey detailed results **154**

C MansOS **156**
C.1 MansOS Execution models 156
C.1.1 Event-based execution 157
C.1.2 Threaded execution 158
C.1.2.1 Cooperative proto-threads 158
C.2 MansOS networking protocol stack 162
C.2.1 Physical layer 162
C.2.2 MAC layer 162
C.2.2.1 Network layer 163

D OOMOS **165**
D.1 Object-oriented operating system advantages 165
D.2 OOMOS source code examples 166

List of Figures

1.1 A typical wireless sensor node architecture . . . 13
1.2 TMote Sky . . . 15

2.1 Abstractions for sensor networks . . . 21
2.2 MansOS components and abstraction layers . . . 25
2.3 MansOS architecture . . . 26

3.1 Distribution function of mote count in surveyed deployments . . . 38
3.2 Sensors used in deployments . . . 39
3.3 Sensor sampling rate used in deployments . . . 40
3.4 Number of kernel level software services used in deployments . . . 46
3.5 Number of application layer software tasks used in deployments . . . 46

5.1 Tactile ship bridge alarm system architecture . . . 75
5.2 Tactile belt prototype . . . 76
5.3 Tactile belt architecture . . . 77

6.1 OOMOS MCU class diagram with new MCU supported . . . 91
6.2 Test application program code size comparison in MansOS, OOMOS, Contiki and TinyOS . . . 95
6.3 Test application static RAM size comparison in MansOS, OOMOS, Contiki and TinyOS . . . 95
6.4 ADC sampling performance of MansOS, OOMOS, Contiki and TinyOS . . . 98
6.5 Radio transmission throughput dependance of packet size, packets/sec . . . 99
6.6 Radio transmission throughput dependance of packet size, KiBytes/sec . . . 100
6.7 Absolute received packet count dependance on packet size . . . 102
6.8 OOMOS code size reduction by excluding unused components . . . 103
6.9 MansOS, OOMOS, Contiki and TinyOS size comparison, lines of source code 106
6.10 MansOS, OOMOS, Contiki and TinyOS size comparison, file count . . . 106
6.11 MansOS, OOMOS, Contiki and TinyOS code categorized . . . 107

6.12 OOMOS, MansOS, Contiki and TinyOS reusability, lines of code 108
6.13 Zolertia Z1 platform in OOMOS, MansOS, Contiki and TinyOS, lines of code 108
6.14 Device driver code in OOMOS, MansOS, Contiki and TinyOS, lines of code 110
6.15 Device driver file count in OOMOS, MansOS, Contiki and TinyOS 110
6.16 Object-oriented OOMOS compared to procedural MansOS, lines of code . 111
6.17 Object-oriented OOMOS compared to procedural MansOS, file count . . . 112

C.1 Flowchart of MansOS application using event-based execution model . . . 157
C.2 Flowchart of MansOS application using preemptive threads 159
C.3 Flowchart of MansOS application using cooperative proto-threads 160

List of Tables

1.1 Typical mote resource limits . . . 16

3.1 Deployments: general information . . . 33

4.1 WSN software design rules proposed by the author . . . 54
4.2 Addressing WSN problems by design rules . . . 61

5.1 Existing OS conformance to proposed design rules . . . 68

6.1 Packet reception rate (PRR) dependance on packet size . . . 101
6.2 OOMOS code and RAM size optimization by excluding unused components 102
6.3 Zolertia Z1 platform source code size in OOMOS, MansOS, Contiki and TinyOS . . . 109

A.1 Deployments: deployment state and attributes . . . 134
A.2 Deployments: sensing . . . 135
A.3 Deployments: lifetime and energy . . . 137
A.4 Deployments: used motes and radio chips . . . 138
A.5 Deployments: used microcontrollers . . . 140
A.6 Deployments: external memory . . . 141
A.7 Deployments: sensor and user interface . . . 141
A.8 Deployments: communication . . . 142
A.9 Deployments: communication media . . . 144
A.10 Deployments: network . . . 145
A.11 Deployments: networking protocol stack . . . 146
A.12 Deployments: used operating system and middleware . . . 147
A.13 Deployments: software level tasks . . . 148
A.14 Deployments: task scheduling . . . 150
A.15 Deployments: time synchronization . . . 151
A.16 Deployments: localization . . . 151

A.17 Deployments: remote access 152

B.1 Sensor network motes designed in years 2010 and 2011 154
B.2 Sensor network motes 2010-2011: MCU and memory 155
B.3 Sensor network motes 2010-2011: radio communication 155

List of source codes

C.1 MansOS socket application example 163
D.1 OOMOS interface example 166
D.2 OOMOS UART interface 166
D.3 OOMOS radio interface 166
D.4 OOMOS CC2420 device driver (partial) 167
D.5 OOMOS TelosB platform initialization (partial) 168
D.6 OOMOS interface for abstract hardware platform 168
D.7 OOMOS protocol interface and base class prototypes 169

Glossary

3G 3rd Generation mobile telecommunications, a generation of standards for mobile phones and mobile telecommunication services

6lowPAN IPv6 over Low power Wireless Personal Area Networks

AC Alternating current

ACM Association for Computing Machinery - global society of educational and scientific computing

Ad Hoc A solution designed for a specific problem or task, non-generalizable

ADC Analog-to-Digital Converter - A hardware module converting analog input signal to digital output signal

ANSI American National Standards Institute - A standardization organization in the United States of America

AODV Ad-hoc On-demand Distance Vector - a routing protocol for ad-hoc networks

API Application Programming Interface - a software library that includes specification for data structures and routines used to interface with a software system

Bluetooth RFCOMM A simple set of transport protocols providing emulated RS-232 serial ports over Bluetooth connection

CDF Cumulative Distribution Function - a function describing the probability that a random variable X with a given probability distribution will be found at a value less than or equal to x

CPU Hardware within a computer system which carries out the instructions of a computer program by performing the basic arithmetical, logical, and input/output operations of the system

CRC Cyclic Redundancy Check - an error-detecting code commonly used in digital networks and storage devices to detect accidental changes to raw data

CSMA Carrier Sense Multiple Access - a probabilistic media access control approach where each node verifies absence of other traffic before transmitting its own data over a shared communication medium

Duty cycling Algorithm of system operation, where part of time is spent in low-power suspend mode

EMF Electro-Magnetic field - a physical field produced by electrically charged objects

FAT16/FAT32 A legacy file system format initially developed for personal computers

FIFO First In First Out - a principle where set of objects are handled in the same order as they were registered in the set

GPIO General Purpose Input/Output - a generic pin on an integrated circuit whose behavior can be programmed by the user at run time

GPS Global Positioning System - a space-based satellite navigation system that provides location and time information in all weather, anywhere on or near the Earth, where there is an unobstructed line of sight to four or more GPS satellites

IDE Integrated Development Environment - a software application that provides comprehensive facilities to computer programmers for software development

IEEE International Electrical and Electronics Engineers - international association of technology professionals

IoT Internet of Things - an Internet-like network formed of uniquely identifiable objects

IPv6 Internet Protocol version 6 - the latest version of IP protocol, addresses the problem of IPv4 protocol's insufficiently large address space

ISO International Organization for Standardization - An international standardization organization

ISO OSI Network protocol stack architecture proposed by ISO organization

ITS Intelligent Transportation Systems - initiative to create more efficient transportation systems by integration of advanced information technologies

KiB Kibibyte, unit for quantifying digital information, equals to 1024 bytes

LCD Liquid-Crystal Display - flat panel display technology that uses light modulating properties of liquid crystals

LED Light-Emitting Diode - a two-lead semiconductor light source that resembles a basic diode and emits light

LTE Long Term Evolution, marketed as 4G LTE - a standard for wireless communication of high-speed data for mobile phones and data terminals

MAC Medium Access Control: a layer in network protocol stack providing efficient transmission medium distribution between multiple network nodes using the medium

MCU Microcontroller - small computer on a single integrated circuit containing a processor core, memory, and programmable input/output peripherals

MiB Mibibyte, unit for quantifying digital information, equals to 1024 kibibytes

MMC MultiMediaCard - a non-volatile flash memory card standard, superseded by Secure Digital card format

Mote Wireless sensor node (hardware platform)

NFC Near Field Communication - a set of standards for mobile devices to establish radio communication with each other by touching them together or bringing them into proximity

OOMOS Object-oriented MansOS operating system

OOOS Object-Oriented Operating System - an operating system that is developed and interfaced using object-oriented programming

OOP Object-Oriented Programming - a programming paradigm that is built around data entities with fields and methods, called objects, as the central concept

Optimized code program source code that uses specific constructs or algorithms, and omits parts of generic implementation to improve performance and resource efficiency for a particular hardware platform or application

OS Operating System - a set of software that manages hardware resources and provides common user interface

PC Personal Computer - a general-purpose computer targeted to be operated directly by end-user

PDA Personal Digital Assistant - a mobile device that functions as a personal information manager

PHY Physical Layer - the lowest layer in ISO OSI protocol stack

PRR Packet Reception Ratio - the ratio between received packets and total number of transmitted packets

QoS Quality of service - intents to improve quality of data transmission to satisfy certain requirements

RAM A form of computer data storage for quick access in random order

Reusable code program source code implemented in a platform-independent manner that can be compiled for multiple platforms

RFID Radio-Frequency IDentification - technology that uses wireless non-contact radio-frequency electromagnetic fields to transfer data, for the purposes of automatically identifying and tracking tags attached to objects

RS-232 The traditional name for a series of standards for serial binary single-ended data and control signals connecting between data terminal equipment and data circuit-terminating equipment

SD Secure Digital - a non-volatile memory card format

Sink oriented communication Type of communication where the data flow is mainly directed towards a single network node

SoC System on chip - an integrated circuit, that integrates all components of a computer or other electronic system into a single chip

SPI Serial Peripheral Interface bus - synchronous serial data link providing simultaneous transmit and receive operations

SQL Structured Query Language - a human-readable format for specification of queries to relational data bases

TDMA Time Division Multiple Access - a media access control approach where each node transmits and receives data only in specific time slots in a previously agreed schedule

TinyOS Tiny Operating System - a component-based operating system for wireless sensor networks, developed using the event-driven nesC language - a C programming language dialect with specific component wiring and synchronization constructs, optimized for resource constrained devices

TinyOS AM Tiny Active Message - a communication protocol stack and message format used in TinyOS operating system

UART Universal Asynchronous Receiver/-Transmitter - a hardware module translating data between parallel and serial formats

VM Virtual Machine - a software-based emulation of a computer

WiFi Wireless local area network products that are based on IEEE 802.11 standards, widely used as a synonym for consumer WLAN access

WSN Wireless Sensor Network - a network of nodes with sensors and wireless communication devices capable of measuring environmental phenomena and reporting the data to one or several locations

1 Introduction

Environmental scientists, biologists, geologists and other researchers and industry professionals are interested in measuring a variety of parameters and phenomena of our planet. Wireless sensor networks (WSNs) is a paradigm of measuring and event detection in the surrounding environment. It is a tool for smart sensing of our planet, having wide variety of applications, including wild animal monitoring [1], remote island flora inspection [2], volcano eruption prediction [3], interactive dance music generation [4], restricted area monitoring [5], and battlefield surveilance [6], among others.

Sensor network consists of multiple nodes (also called *motes*), which sense the environment and exchange data with each other. In most cases, the gathered information is transferred to a central collection point, called *base station* or *sink*.

A sensor node consists of power (typically batteries), sensing, computation and communication parts, sensors, micro-controller (MCU) and a wireless transceiver (Figure 1.1). External flash memory is optional and is used to store large collected data streams.

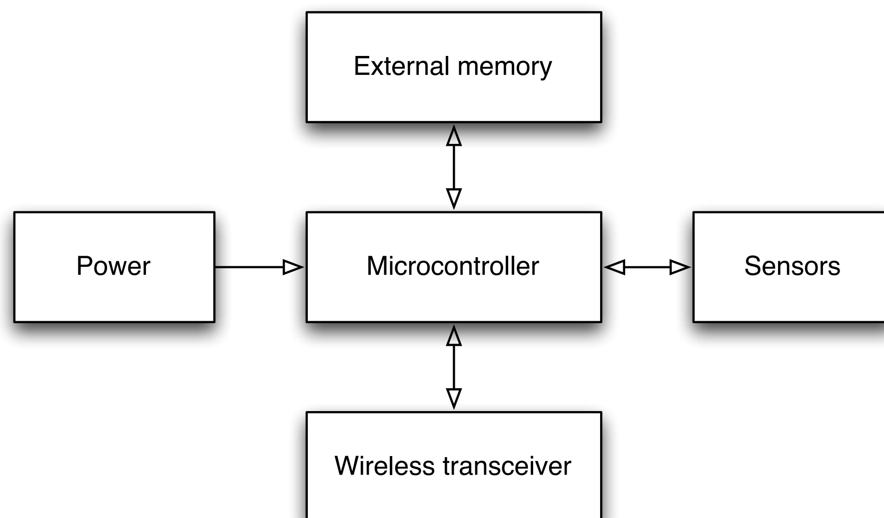

Figure 1.1: A typical wireless sensor node architecture - power, microcontroller, wireless transceiver, sensors and external memory (optional)

Considering WSN peculiarities, specific solutions have been proposed. Low-power MCU [7] and radio [8] chips are developed, suitable communication media are used [9], available infrastructure is used for communication support [10], physical radio propagation phenomena are used [11]. New communication protocols are developed for physical layer [12], Media access control (MAC) [13], network [14] and transport layers [15].

1.1 Project experience

The author has participated in several research projects related to wireless sensor network programming, which have helped to get better understanding of common problems in WSN application design. The author's experience has been developed during participation in the following projects:

1. Development of MansOS [16, 17]: an operating system for wireless sensor networks with the focus of providing common Unix-like environment for C programmers, rapid familiarization, prototyping and porting to new hardware platforms.

2. Design of low-power, delay-tolerant sensor network solutions for wild animal monitoring [18].

3. Microclimate monitoring in fruit orchards for precision-agriculture solutions [19].

4. Development of flexible hardware platforms for WSN prototyping [20].

5. Vehicular sensor network [21] applications in urban scenarios: pothole detection [22, 23, 24, 25] and cooperative driving [26, 27].

6. Smart road infrastructure for increased traffic safety [28].

7. Detection of human gestures with vision-based sensor networks [29].

In addition, the author has performed extensive WSN deployment survey analyzing different aspects of WSN applications. WSN software development design rules, which are formed based on the analysis of data found in the survey, form a central part of this thesis, and have been published in a scientific article [30].

1.2 Scope and motivation

Sensor nodes typically are match-box size embedded devices (Figure 1.2). Such devices are called *motes*. Other networks of devices with sensing and communication capabilities can also be considered sensor networks, for example vehicular sensor networks with car on-board computers [22] or human-centric networks of smartphones [31]. However, these kinds of networks use devices significantly different from motes, and therefore different software abstractions could be more appropriate. In this work, the main focus is on networks consisting of *mote* devices.

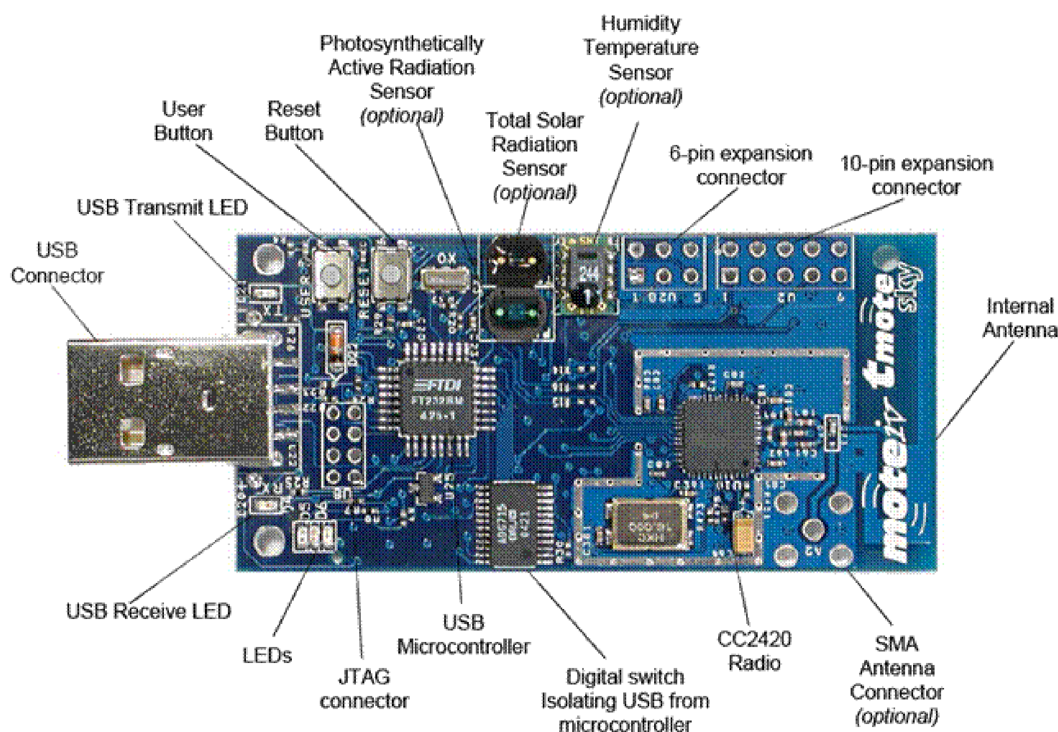

Figure 1.2: TMote Sky - a typical match-box size wireless sensor node (also called *mote*). Picture from [32].

Sensor network design and programming contains a set of challenges to solve:

1. Limited energy budget and energy efficiency is the main challenge of sensor networks. High-capacity batteries and energy harvesting methods [33] have to be combined in an efficient way, using energy buffering and duty cycling.

2. Small size and low-power computation (see Table 1.1) implies limited central processing unit (CPU), random-access memory (RAM) and flash memory resources, which require highly effective software abstractions and low-level hardware control.

3. Standard communication hardware and protocol stacks used in Internet servers and desktop computers are not suitable for sensor networks, due to low-power requirements and duty cycling.
4. Different platforms are often used during evolution of the projects. System design requires rapid prototyping, including hardware and software tool support.
5. Embedded systems are event-driven, while most desktop programming languages provide sequential programming paradigms. Therefore easily-adoptable software abstractions must be provided to WSN programmers, without requirement of long learning process.
6. Sensor nodes are often deployed in the wild, industrial or other open and harsh environment, requiring additional effort for packaging problem solution.
7. Wireless communication has remarkable irregularities and disturbances, which must be mitigated by both hardware and software methods.
8. Despite the unfriendly environment, sensor networks should be fault tolerant and self-adaptive.
9. Deployment site is often hardly reachable for physical hardware inspection and maintenance. Remote real-time management support is therefore desirable.

Table 1.1: Typical mote resource limits - CPU performance and memory amounts are serious constraints for software design

Resource	Typical amount
CPU	1 - 20MHz
RAM memory	256B - 10KiB
Non-volatile flash memory	40 - 128KiB
Optional external flash memory	1MiB - 4GiB

Wireless sensor network research has evolved over the last decade and has reached a state where standardization becomes essential for interoperability between different hardware and software solutions. Although parts of the WSN solutions are standardized, such as communication protocols (802.15.4 standard [34]), a common methodology for WSN software development is still missing. WSN designers face typical problems during software development.

Therefore central thesis of this work states: *a common methodology is required for wireless sensor network software development.* Such methodology would foster efficient new solution design and serve as a tool for existing software evaluation and identification of improvements.

The rest of this work describes the process of identification of common WSN problems, WSN design rule proposal and evaluation in different aspects: existing software assessment and new software design. These rules form a basis for common WSN software development methodology. Although standardization is a slow and complex process beyond the scope of single person's competence, these rules can have impact towards establishment of common WSN software development standards and protocols.

1.3 Contribution of the thesis

The author's main contribution in this thesis includes:

1. Analysis of 40 sensor network deployments described in the research literature. As a result the critical and recurring WSN properties were distilled.
2. Identification of common WSN design problems that identify the challenges based on critical WSN properties and user requirements.
3. Introduction of a WSN software development methodology in the form of 25 design rules and analysis of their mapping to underlying problems.
4. Evaluation of the proposed design rule impact on existing WSN software improvement. Design rules are shown as a tool for existing system comparison, drawback identification and future direction sketch. The evaluation consists of three parts:
   (a) Improvements to the analyzed deployment set showing design rule applicability in general, for WSN users.
   (b) Existing operating system conformance to proposed rules and suggestions for OS improvement. Design rules are shown as an important tool for WSN OS developers. This evaluation includes the author's participation in the development and improvement analysis of MansOS: a portable operating system (OS) for sensor networks.
   (c) A wearable sensor network use-case scenario - assessment of prototype implementation and suggestions for future work. This part shows more detailed improvement of a particular WSN deployment in terms of network lifetime and network coverage.

5. In addition, the author has developed a new WSN operating system, (called Object-Oriented MansOS or OOMOS) according to the rules. This part of the thesis shows design rules as a valuable tool in early stages of WSN OS design and implementation.

1.4 Related work

For WSN requirement summary and trend inference we first have to survey existing WSNs and establish a taxonomy. Numerous researchers have surveyed and described sensor network characteristics and challenges. In her book Anna Hac describes sensor networks in general, including typical challenges [35]. Hill et.al. describe WSN hardware platforms [36]. Tilak et al. propose to categorize sensor networks based on different criteria [37]. Mottola and Picco propose another taxonomy focusing on programming aspects [38]. The author of this thesis also proposes a WSN taxonomy that is based on the author's experience and summary of multiple survey articles, see Appendix A.1.

Metric definition is also an important task. Beutel proposes metrics for WSN hardware platforms [39]. The author of this thesis used subset of these metrics for deployment analysis in Chapter 3. However, the author adds significantly more metrics in the survey, see tables in Appendix A.2.

Romer and Mattern have analyzed WSN deployments and assessed the WSN design space based on application characteristics [40]. This thesis includes similar deployment analysis approach. Handziski et.al. have analyzed WSN challenges and come to conclusions similar to the author's: standardizations and unified methodologies are required [41]. Handzinski is proposing suggestions for handling the challenges, without formalization. Jason Hill's thesis [42] is the closest effort to this thesis. He analyzes WSN system architecture, describes constraints and challenges. Hill substantiates TinyOS design choices with qualitative suggestions based on the identified challenges. The author of this thesis takes a step further and proposes specific design rules based on a deployment survey. The WSN survey is performed similarly to previous work, yet with more detail and more formalized outcomes: the proposed design rules.

Different approaches are possible in WSN design and specification. Several researchers have proposed algorithms and formulas to optimize WSN communication. Mhatre and Rosenberg propose an algorithm how to choose between different communication approaches [43]. Olariu and Stojmenovic describe formulas how to calculate energy depletion dependence on network topology and optimal transmission power [44]. Stojmenovic et.al. describe design guidelines for WSN routing protocols [45]. Oppermann and Peter

propose a framework to transform informal end-user requirements to technical specifications [46]. It tries to solve communication problem between different WSN user groups: end-users and engineers.

In contrast, the author of this thesis focuses on optimizing software development process, not communication protocols or social communication problems. He extracts technical WSN deployment characteristics based on the information available. In some cases it is not possible to gather quantitative information. Qualitative discussion is used in such situations, including proposed design rule evaluation in Chapter 5 and Chapter 6. Nevertheless, to the best of the authors knowledge, this work proposes the most comprehensive and formalized set of design rules for WSN software development.

1.5 Summary and thesis outline

Wireless sensor network software development includes set of challenges due to distributed nature, resource limitations and environmental constraints of the networks. A common methodology is important for standardization, interoperability and component reuse across different WSN applications. This thesis analyzes different WSN software abstractions and shows that operating systems are an important part of WSN software (Chapter 2). An extensive WSN deployment survey will be presented in Chapter 3 analyzing common trends and requirements of WSN applications. Subsequently, typical WSN problems will be identified (Section 4.1) and design rules for WSN software development will be proposed (Section 4.2). Relation between problems and rules will be analyzed in Section 4.3. The proposed design rules will be evaluated in different aspects in the two following chapters: Chapter 5 and Chapter 6. First, rule impact on analyzed WSN deployments will be discussed in Section 5.1. Second, existing WSN operating system conformance to proposed rules will be analyzed in Section 5.2. The author participated in MansOS operating system development, therefore special attention will be devoted to design rule impact on MansOS evolution (Section 5.2.5). Third, a particular WSN use case (where the author also contributed in system design and implementation) will be analyzed by describing its current prototype state, assessing it and proposing directions for improvement, using proposed design rules (Section 5.3). And last, Chapter 6 will be devoted to discussion of design rule applicability during planning and designing OOMOS: a new, object-oriented OS for sensor networks, built by the author on principles of MansOS and according to proposed design rules. Details of OOMOS implementation will be described in Section 6.3. OOMOS will be evaluated in terms of portability and performance in Section 6.4. Future work on OOMOS improvement according to proposed design rules

will be discussed in Section 6.5. The thesis will be completed with conclusions on results and future work in Chapter 7.

2 Sensor network software abstractions

Sensor networks can be programmed (or tasked) in multiple ways, using different abstractions (Figure 2.1). The choice of abstraction to use is up to user and depends on application requirements and user skills.

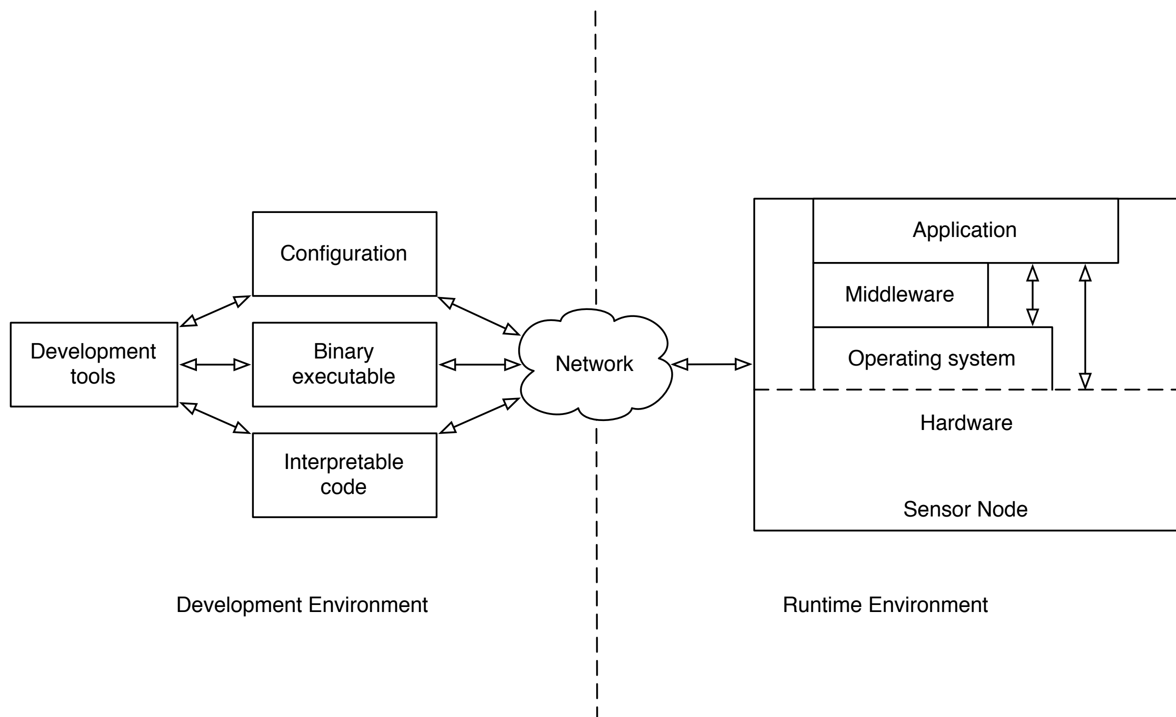

Figure 2.1: Abstractions for sensor networks - Application can be written using custom code, on-top of operating system or using additional middleware. The whole network can be reprogrammed or re-tasked remotely

Two different environments must be distinguished:

a) Software abstractions in runtime environment - different layers of tasks or operations executed on the sensor nodes. An example: a web server (application) running on a

sensor node that uses database (middleware) for sensor data, and this database is built on top of TinyOS (operating system).

b) Programming abstractions in development environment abstractions - tools and methods used to provide better user experience in the process of sensor node programming. Users can program sensor nodes in C/C++, build the code on top of operating systems and virtual machines. The produced source code can be directly compiled to binary code for sensor node microcontroller, it can also undergo multiple layers of translation and compilation to different interpretable middleware formats. From user perspective there is no difference between program source code pre-compilation to binary image in the development environment and its interpretation or translation on the sensor node during deployment.

The following abstractions are used in the runtime environment:

1. Operating system - interface for efficient access to hardware (Section 2.1).

2. Middleware - an intermediate layer providing access in a different, domain-specific approach (Section 2.2).

3. Application - user created program for task-specific operations.

Each next software abstraction on the sensor node increases usability and convenience for the user (or programmer), while sacrificing certain amount of performance and flexibility. The most efficient way would be to write program code directly in machine language or assembly for each application individually - all of hardware features are available in an efficient way, using this abstraction. However, such approach is available only to embedded system programming experts, requires lot of time and effort, and is more prone to software bugs. Therefore for each application and user type an appropriate programming abstraction must be chosen, providing required development convenience and resource efficiency.

It is important to remember, that sensor node is just one part of sensor network. It can be considered a peripheral device of the system. The whole network includes also routers, gateways (which can use the same sensor node hardware) and base stations, usually connected to a personal computer (PC) for data collection, analysis, post-processing and visualization. A complete sensor network programming includes software tools for all the listed network components.

2.1 Operating systems

The lowest software abstraction in runtime environment, most closely to hardware, is operating system. It provides device drivers and efficient user interface for application, higher abstraction and service development. Often user applications are built right on top of operating system, without any middleware services. There are multiple reasons why efficient access to hardware may be required:

1. Sensor network hardware platforms often have very limited computation resources.
2. High-frequency, accurately timed sensor sampling is required for certain applications.
3. Specific sensors or advanced chip features, not provided in hardware-independent layer, are sometimes required.

The goal of a conventional PC-based operating system is to provide convenient and efficient interface between users and hardware resources. The same goal holds for wireless sensor network OS. However, sensor networks possess different hardware platforms, applications and users. Therefore different strategies are used to reach the same goals effectively.

In contrast to conventional computers, sensor networks usually use an OS as a set of libraries and functions which are linked together with user application code in a single firmware image. It is in some occasions hard to separate user application and OS kernel. For resource-efficiency reasons separate kernel thread and system call interface may be substituted by direct resource access to the user application code. However, some operating systems, such as Contiki [47], use dynamic application linking and process loading.

Multiple wireless sensor network operating systems have been proposed previously by the research community, most widely known are TinyOS [48], Contiki [47], MansOS [16], LiteOS [49], Mantis [50] and Enix [51].

The following subsections describe the listed operating systems. MansOS is described in more detail, compared to others. Author has actively participated in MansOS development since 2008 and it will be used as a use case to show how an OS can be improved by applying proposed design rules in its development.

2.1.1 MansOS

This section describes operating system MansOS [17], developed by research team, where the author of this thesis is participating. MansOS[1] is modular and portable WSN operating system that provides programming interface in plain C language, environment and concepts familiar to Unix programmers. It's main purpose is to serve as a *sandbox* for WSN software solution development and experimentation.

The author has used experience in MansOS development as one source of knowledge during writing of this thesis. Therefore it is not surprising, that MansOS conforms to majority of proposed design rules. Nevertheless, as the evaluation will reveal in Section 5.2.5, there is space for improvement. This section summarizes important aspects of MansOS, more detailed description in [16].

MansOS design is based on a set of principles, with the ease of WSN application programming and portability in mind:

- *Source code abstractions*: source code is separated in four parts: chip-specific, MCU architecture-specific, platform-specific, and platform-independent application programming interface (API) (see Figure 2.2). Such separation provides more flexible code development and porting. For example, chip-specific device drivers can be used in multiple platforms, and all platforms may implement the same platform-independent API by calling appropriate chip-specific functions.

- *Modular system* allowing to select among multiple alternative implementations (for example, cooperative of preemptive task scheduling) and switch off unused modules for the particular application, to reduce code size, increase performance and lifetime.

- *Core libraries* provide essential networking protocols and services for WSN applications, such as file system, time synchronization, cyclic redundancy check (CRC) checksum calculation, and random number generation.

- *Unix-like programming using only C language following American National Standards Institute (ANSI) specification.* No language extensions or preprocessors are added to provide familiar environment and avoid confusion.

- *Different scheduling techniques.* MansOS provides three different schedulers: cooperative, preemptive and event-based.

[1] Available at `http://mansos.net`.

- *Simulation on the PC platform.* Before real deployments MansOS provides simulated environment to compile and test application level algorithms on PC platform. Network consisting of multiple nodes can be simulated.
- *Remote management and reprogramming.* To accelerate deployment-time debugging and facilitate reprogramming, run-time interface for data access and reprogramming over-air is provided.
- *Support of popular hardware chips and architectures.* MansOS supports both AVR and MSP430 microcontroller architectures, and popular WSN mote platforms, including TelosB with 802.15.4 compatible radio chip CC2420.

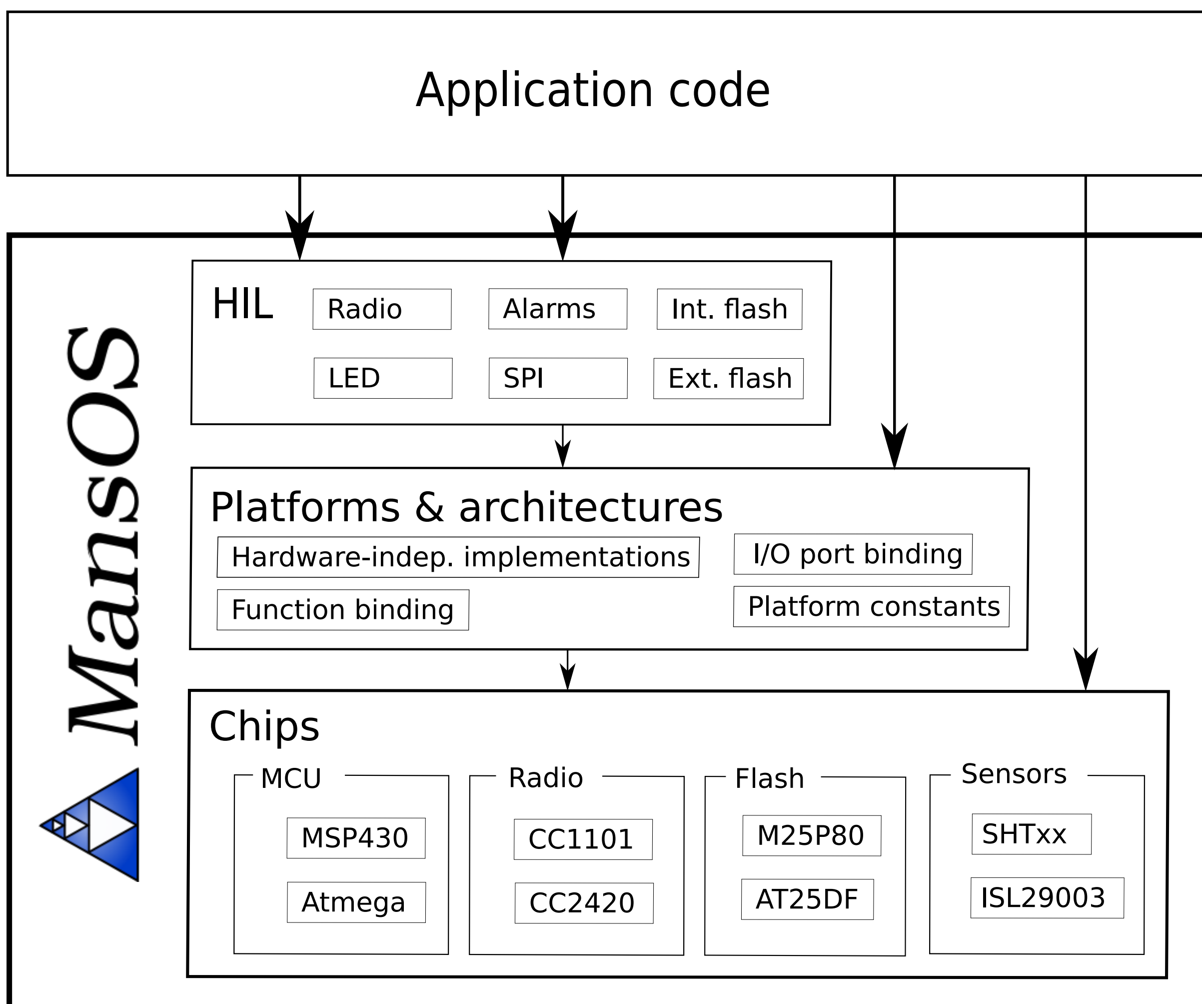

Figure 2.2: MansOS components and abstraction layers - programmers have direct access to all system layers, from chip-specific code to hardware-independent-layer (HIL). Figure from [16]

MansOS architecture is shown in Figure 2.3. The following tasks are executed on the sensor node:

- First, initialization starts with a bootloader. Multiple options are available - a simple bootloader passing execution to kernel initialization, and a more complex bootloader supporting remote node reprogramming.
- Kernel initialization selects the active components, based on hardware available on the particular node, and the configuration at compile time. Calls to device drivers are made.
- A specific time service routing is started that manages time counter and timed event interrupt handling.
- Kernel starts task scheduler which distributes CPU time among kernel and user tasks based on the execution model (see Section C.1 in Appendix for more details on different execution models available in MansOS).
- The only critical process implemented in the kernel space is radio communication (time synchronization is included in packet exchange). The rest of processes are implemented in the user space, depending on application requirements.

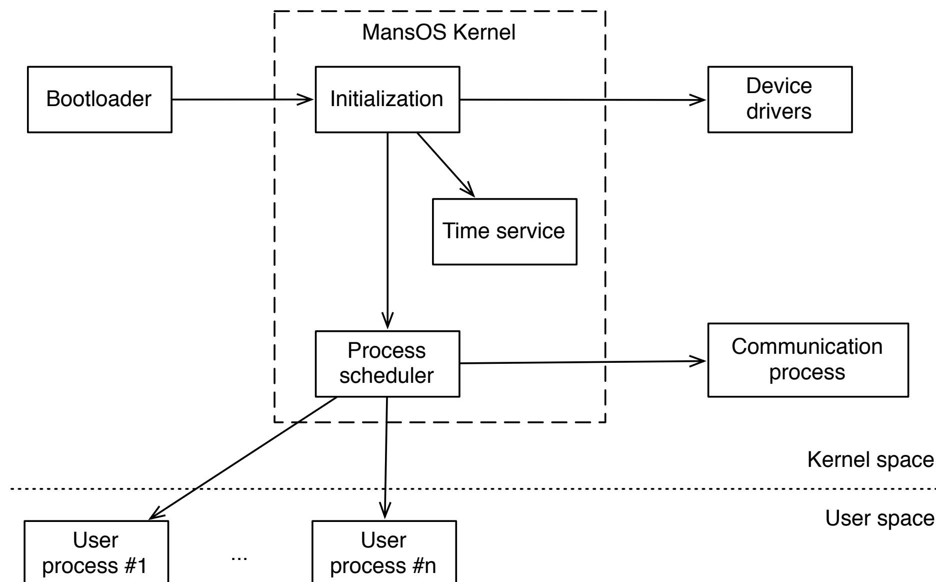

Figure 2.3: MansOS architecture - Bootloader calls kernel code responsible for device driver initialization and process scheduling

MansOS is a flexible WSN operating system, that allows users to access radio communication at multiple International Organization for Standardization (ISO) Open Systems Interconnection (OSI) networking stack layers [52]: physical, data link (MAC), and network. Transport and application layers are not core features of MansOS, yet they may be implemented using external libraries and user application code. Session and presentation OSI layers are usually not used in WSN applications.

MansOS allows interchangeable, independent protocols to be used in all three networking stack layers. For example, the same routing protocol may be used on top of different MAC protocols. MansOS includes simple carrier-sense multiple access (CSMA) MAC and multi-hop routing protocols, with package acknowledgements, and allows developers to customize them, and to implement completely new protocols. See Section C.2 in Appendix for more details on MansOS networking protocol stack.

MansOS features a simple file system that abstracts the physical storage as a number of logical files or streams. Following the MansOS philosophy, the file system interface is synchronous (UNIX-like) and thread-safe. In addition to basic file commands, the system has non-buffering and integrity-checking modes. On the low level, the system is designed for flash chips that have very large segments and don't contain integrated controllers that handle data rewrites and wear leveling.

MansOS is multi-platform in the sense of supporting multiple hardware platforms (Tmote Sky, Arduino, Zolertia Z1 and more) and multiple architectures (MSP430 and Atmel AVR). MansOS provides platform-independent API for analog sensor sampling using analog-to-digital converter (ADC) module, digital communication protocols (Universal asynchronous receiver/transmitter (UART), serial peripheral interface(SPI), inter-integrated circuit (I^2C)) and MCU pin configuration. Therefore sensor, external memory and other peripheral drivers can be designed using platform-independent routines, allowing the same driver to be reused among multiple platforms and applications. The communication protocols are provided in both hardware and software versions using unified API. The version to be used is selected at compile time, allowing to reuse peripheral drivers without modification.

2.1.2 TinyOS

TinyOS is the first operating system designed especially for wireless sensor networks. It is actively supported, well tested and has created a wide contributor and user community, and can be considered *de facto* standard for WSN programming. TinyOS is primarily targeted to sensor network researchers. According to Levis et al [48]:

> The space of networked sensors is novel and complex: we therefore focus on flexibility and enabling innovation, rather then the right OS from the beginning.

Resource-constrained sensor nodes requiring high energy efficiency are event-driven embedded devices. Therefore compact, reactive scheduler is used (core system uses 400 bytes of program memory). Source code is written in *nesC* language, a C dialect with minor modifications, that is processed by a *nesC* parser and pre-compiled into a single C source file. This single file is then compiled into a firmware image and takes advantages of static compiler optimizations.

TinyOS is a highly modular system, consisting of components, wired together using specified interfaces. Each component provides a particular service and interfaces describe *commands* for starting a service and *events* for signaling completion of a service routine. Inside components low-priority tasks are scheduled, using non-preemptive, run-to-completion first in first out (FIFO) task queue. High-priority event handlers are used for time-critical section execution. Optional preemptive scheduler can be used, implemented as an add-on, called TOSThreads [53].

2.1.3 Contiki

Contiki is a lightweight operating system with support for dynamic loading and replacement of individual programs and services. It is built around an event-driven kernel and provides optional preemptive multithreading [47]. Contiki is written in C language and has been ported to a number of platforms, including TelosB and MicaZ, having different CPU architectures: Atmel AVR, Texas Instruments MSP430 and others. Bundled application examples are extensible and provide easy learning and experimentation interface for novice users.

The only abstractions provided by Contiki kernel are CPU multiplexing and dynamic program and service loading. Additional abstractions are provided by libraries with full access to underlying hardware. Loadable programs are implemented, using modified binary format containing relocation information and performing runtime relocation.

Communication stack is composed of services. Therefore, each layer is replaceable in runtime and multiple communication stacks are loadable simultaneously.

Contiki is, perhaps, the most widely used TinyOS alternative, providing more classical sequential programming approach and rich service library. One argument supporting this statement is Contiki developer activity in the forum with several hundred emails being discussed each month [54]. From design perspective, Contiki has potential for change, including improvements in platform-independency increase.

2.1.4 LiteOS

LiteOS is a multi-threaded operating system that provides Unix-like abstractions for wireless sensor networks [49]. It offers hierarchical file system, remote shell, dynamic application loading, preemptive scheduler for multithreaded applications and and object oriented programming language LiteC++ - a subset of C++. LiteOS has been implemented on MicaZ and Iris mote platforms, both with AVR microcontrollers.

LiteOS utilizes a specific binary image format, where all memory-dependent instructions are modified, using mathematical model, which calculates relative addresses from the statically compiled ones.

LiteOS demonstrates a list of interesting Unix-like concepts integrated into a sensor network OS. MansOS operating system, described in Section 2.1.1, was initially started as a branch of LiteOS with the same goals to provide a flexible and familiar programming WSN environment for Unix-users.

2.1.5 Mantis

Mantis is a multithreaded cross-platform embedded operating system for wireless sensor networks, supporting complex tasks such as compression, aggregation and signal processing, implemented in a lightweight RAM footprint that fits in less than 500 bytes of memory, including kernel, preemptive scheduler, and network stack [50]. Mantis is implemented on multiple platforms, including PCs and personal digital assistants (PDAs), allowing to create hybrid networks consisting of real sensor nodes and virtual ones, simulated on one or multiple PCs. Written in C language, Mantis OS translates to a separate API on the PC platform.

Device drivers are implemented in Mantis similarly to unix device files. Additional abstraction layer is implemented in the kernel, providing blocking call interface for external event waiting. Remote shell is implemented, providing access to mote program and data memory. Therefore, remote reconfiguration and partial reprogramming is possible in theory, yet automated techniques have not been developed.

2.1.6 Arduino

According to [55]: "Arduino is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for artists, designers, hobbyists and anyone interested in creating interactive objects or environments."

Arduino has a significantly different ideology and scope compared to traditional WSN operating systems. Arduino is both hardware and software solution for generic-purpose embedded systems. The main advantage of Arduino is it's huge community that contributes with different libraries and application examples, shared ideas and completed projects. Although in it's core Arduino has a very limited functionality, it's extensions and examples make it powerful.

However, neither Arduino's hardware nor software components are designed for low-power requirements of WSNs. A typical Arduino application has 100% duty cycle and a single thread of execution. The core hardware platform having only microcontroller consumes 25mA in active mode, that is more than 1000% of TMote Sky sensor node's energy consumption in radio-off mode and more than TMote Sky node's consumption when radio is active.

Nevertheless, this platform is practical for rapid prototyping and WSN scenarios where energy consumption is not critical (either short lifetime is expected or stable power source is available). Compared to Arduino, high-performance platforms, such as Raspberry Pi [56] or Odroid [57], running Linux can provide richer software environment and more on-board data processing power. Yet the advantages of Arduino are simplicity and low price.

2.2 Middleware

On top of operating systems, middleware can be used in the runtime environment to provide programmers more convenient or specialized access to sensor network resources. In addition to runtime middleware different middleware tools can also be part of development environment providing translations and compilations outside sensor nodes.

Examples of middleware abstractions in the runtime environment include:

- Virtual machines (VMs) providing more comfortable or traditional programming environment, such as Java virtual machines for WSNs [58, 59, 60].

- Virtual machines providing more compact code size therefore increasing energy efficiency of remote sensor network reprogramming and retasking [61, 62].

- Macro-programming abstractions with communication neighborhood and data aggregation as programming primitives [63, 64].

- Query languages, including structured query language (SQL)-like languages, treating the whole network as a distributed database, for users with database interaction experience [65, 66].

Development environment middleware examples include high-level declarative language interfaces, providing English-like network tasking and configuration, for field experts not familiar with computer programming [67].

Operating systems for sensor networks are suitable for user group with advanced level programming expertise, basic embedded hardware knowledge and understanding of wireless networking principles. To make sensor networks available as a tool for wider user rage, operating systems must support creation of additional middleware layers on top of them. Middleware can significantly decrease complexity of sensor node resource management and network connectivity control by providing domain and application specific programming paradigms.

2.3 Summary

Wireless sensor networks can be programmed using different software abstractions. One very important part of the abstraction hierarchy is operating systems. There are significant differences between WSN and desktop operating systems due to specifics of WSN hardware and environment. Several WSN operating systems have been proposed in previous work, including MansOS operating system where the author has also contributed. Different middleware solutions can be used on-top of WSN operating systems to provide different, usually more specific and simple, programming interface.

It is important to have a unified methodology for development of WSN operating systems and middleware. Establishment of such methodology is an important step towards efficient WSN solutions, interoperability and standardization.

This chapter has summarized different WSN software solutions. Chapter 3 will describe survey of different WSN deployments and definition of a unified methodology, in the form of design rule set, will be provided in Chapter 4.

3 Deployment survey

The goal of software abstractions, and operating systems in particular, is to simplify practical application development and deployment prototyping (Chapter 2 described WSN software abstractions in more detail). Prototyping is also required for communication protocol testing in real-world environment. Therefore this section provides a survey on sensor network deployments with a goal to infer common technical attribute trends. This survey is important for creation of design rule set and common WSN software development methodology design.

3.1 Methodology

Research papers presenting deployments are selected based on multiple criteria:

- Years 2002 up to 2011 have been reviewed uniformly, without any emphasis on a particular year. Deployments before the year 2002 are not considered, as early sensor network research projects used custom hardware, differing from modern embedded systems significantly. Inclusion of such deployments would lead to greater variance of statistical results and less important conclusions in context of near future prediction.
- Articles have been searched using Association for Computing Machinery (ACM) Digital Library [68], Institute of Electrical and Electronics Engineers (IEEE) Xplore Digital Library [69], Elsevier ScienceDirect [70] and SpringerLink databases [71]. Several articles have been found as external references from the aforementioned databases.
- Deployments have been selected to cover the whole taxonomy, described in Appendix (Section A.1).

3.2 Survey results

For each deployment, the best possible parameter extraction was performed. Part of information was explicitly stated in the analyzed papers and web pages, part of it was

acquired by making a rational guess or approximation. Such approximated values are marked with question mark right after the approximated value. Only deployments described in scientific journals and conference proceeding are included in the survey, web news pages are not considered.

General deployment attributes are shown in Table 3.1. Each deployment has a codename assigned. It will be used to identify each article in the following tables.

Multiple parameters are analyzed for each of the considered WSN deployments. For presentation simplification, these parameters are grouped and each group is represented as a separate subsection.

Table 3.1: Deployments: general information

Nr	Codename	Year	Title	Class	Description
1	Habitats [72]	2002	Wireless sensor networks for habitat monitoring	Habitat and weather monitoring	One of the first sensor network deployments, designed for bird nest monitoring on a remote island
2	Minefield [73]	2003	Collaborative Networking Requirements for Unattended Ground Sensor Systems	Opposing force investigation	Unattended ground sensor system for self healing minefield application
3	Battlefield [74]	2004	Energy-Efficient Surveillance System Using Wireless Sensor Networks	Battlefield surveillance	System for tracking of the position of moving targets in an energy-efficient and stealthy manner
4	Line in the sand [6]	2004	A line in the sand: a wireless sensor network for target detection, classification, and tracking	Battlefield surveillance	System for intrusion detection, target classification and tracking
5	Counter-sniper [75]	2004	Sensor Network-Based Counter-sniper System	Opposing force investigation	An ad-hoc wireless sensor network-based system that detects and accurately locates shooters even in urban environments.
6	Electro-shepherd [76]	2004	Electronic shepherd - a low-cost, low-bandwidth, wireless network system	Domestic animal monitoring and control	Experiments with sheep GPS and sensor tracking
7	Virtual fences [77]	2004	Virtual fences for controlling cows	Domestic animal monitoring and control	Experiments with virtual fence for domestic animal control
8	Oil tanker [78]	2005	Design and Deployment of Industrial Sensor Networks: Experiences from a Semiconductor Plant and the North Sea	Industrial equipment monitoring and control	Sensor network for industrial machinery monitoring, using Intel motes with Bluetooth and high-frequency sampling
9	Enemy vehicles [79]	2005	Design and Implementation of a Sensor Network System for Vehicle Tracking and Autonomous Interception	Opposing force investigation	A networked system of distributed sensor nodes that detects an evader and aids a pursuer in capturing the evader

...

Table 3.1 – continued

Nr	Codename	Year	Title	Class	Description
10	Trove game [80]	2005	Trove: a Physical Game Running on an Ad-Hoc Wireless Sensor Network	Child education and sensor games	Physical multiplayer real-time game, using collaborative sensor nodes
11	Elder RFID [81]	2005	A Prototype on RFID and Sensor Networks for Elder Healthcare: Progress Report	Medication intake accounting	In-home elder healthcare system integrating sensor networks and RFID technologies for medication intake monitoring
12	Murphy potatoes [82]	2006	Murphy Loves Potatoes Experiences from a Pilot Sensor Network Deployment in Precision Agriculture	Precision agriculture	A rather unsuccessful sensor network pilot deployment for precision agriculture, demonstrating valuable lessons learned
13	Firewxnet [83]	2006	FireWxNet: A Multi-Tiered Portable Wireless System for Monitoring Weather Conditions in Wildland Fire Environments	Forest fire detection	A multi-tier WSN for safe and easy monitoring of fire and weather conditions over a wide range of locations and elevations within forest fires
14	AlarmNet [84]	2006	ALARM-NET: Wireless Sensor Networks for Assisted-Living and Residential Monitoring	Human health telemonitoring	Wireless sensor network for assisted-living and residental monitoring, integrating environmental and physiological sensors and providing end-to-end secure communication and sensitive medical data protection
15	Ecuador Volcano [3]	2006	Fidelity and Yield in a Volcano Monitoring Sensor Network	Volcano monitoring	Sensor network for volcano seismic activity monitoring, using high frequency sampling and distributed event detection
16	Pet game [85]	2006	Wireless Sensor Network Based Mobile Pet Game	Child education and sensor games	Augmenting mobile pet game with physical sensing capabilities: sensor nodes act as eyes, ears and skin
17	Plug [86]	2007	A Platform for Ubiquitous Sensor Deployment in Occupational and Domestic Environments	Smart energy usage	Wireless sensor network for human activity logging in offices, sensor nodes implemented as power strips
18	B-Live [87]	2007	B-Live - A Home Automation System for Disabled and Elderly People	Home/office automation	Home automation for disabled and elderly people integrating heterogeneous wired and wireless sensor and actuator modules
19	Biomotion [4]	2007	A Compact, High-Speed, Wearable Sensor Network for Biomotion Capture and Interactive Media	Smart user interfaces and art	Wireless sensor platform designed for processing multipoint human motion with low latency and high resolutions. Example applications: interactive dance, where movements of multiple dancers are translated into real-time audio or video

...

Table 3.1 – continued

Nr	Codename	Year	Title	Class	Description
20	AID-N [88]	2007	The Advanced Health and Disaster Aid Network: A Light-Weight Wireless Medical System for Triage	Human health telemonitoring	Lightweight medical systems to help emergency service providers in mass casualty incidents
21	Firefighting [89]	2007	A Wireless Sensor Network and Incident Command Interface for Urban Firefighting	Human-centric applications	Wireless sensor network and incident command interface for firefighting and emergency response, especially in large and complex buildings. During a fire accident, fire spread is tracked and firefighter position and health status is monitored.
22	Rehabil [90]	2007	Ubiquitous Rehabilitation Center: An Implementation of a Wireless Sensor Network Based Rehabilitation Management System	Human indoor tracking	Zigbee sensor network based ubiquitous rehabilitation center for patient and rehabilitation machine monitoring
23	CargoNet [91]	2007	CargoNet: a low-cost micropower sensor node exploiting quasi-passive wakeup for adaptive asychronous monitoring of exceptional events	Good and daily object tracking	System of low-cost, micropower active sensor tags for environmental monitoring at the crate and case level for supply-chain management and asset security
24	Fence monitor [5]	2007	Fence Monitoring Experimental Evaluation of a Use Case for Wireless Sensor Networks	Security systems	Sensor nodes attached to a fence for collaborative intrusion detection
25	BikeNet [92]	2007	The BikeNet Mobile Sensing System for Cyclist Experience Mapping	City environment monitoring	Extensible mobile sensing system for cyclist experience (personal, bicycle and environmental sensing) mapping leveraging opportunistic networking principles
26	BriMon [93]	2008	BriMon: A Sensor Network System for Railway Bridge Monitoring	Bridge monitoring	Delay tolerant network for bridge vibration monitoring using accelerometers. Gateway mote collects data and forwards opportunistically to a mobile base station attached to a train passing by.
27	IP net [94]	2008	Experiences from Two Sensor Network Deployments - Self-Monitoring and Self-Configuration Keys to Success	Battlefield surveillance	Indoor and outdoor surveillance network for detecting troop movement
28	Smart home [95]	2008	The Design and Implementation of Smart Sensor-based Home Networks	Home/office automation	Wireless sensor network deployed in a miniature model house, which controls different household equipment: window curtains, gas valves, electric outlets, TV, refrigerator, door locks

...

Table 3.1 – continued

Nr	Codename	Year	Title	Class	Description
29	SVATS [96]	2008	SVATS: A Sensor-network-based Vehicle Anti-Theft System	Anti-theft systems	Low cost, reliable sensor-network based, distributed vehicle anti-theft system with low false-alarm rate
30	Hitchhiker [97]	2008	The Hitchhikers Guide to Successful Wireless Sensor Network Deployments	Flood and glacier detection	Multiple real-world sensor network deployments performed, including glacier detection, experience and suggestions reported.
31	Daily morning [98]	2008	Detection of Early Morning Daily Activities with Static Home and Wearable Wireless Sensors	Daily activity recognition	Flexible, cost-effective, wireless in-home activity monitoring system integrating static and mobile body sensors for assisting patients with cognitive impairments
32	Heritage [99]	2009	Monitoring Heritage Buildings with Wireless Sensor Networks: The Torre Aquila Deployment	Heritage building and site monitoring	Three different motes (sensing temperature, vibrations and deformation) deployed in a historical tower to monitor its health and identify potential damage risks.
33	AC meter [100]	2009	Design and Implementation of a High-Fidelity AC Metering Network	Smart energy usage	AC outlet power consumption measurement devices, which are powered from the same AC line, but communicate wirelessly to IPv6 router
34	Coal mine [101]	2009	Underground Coal Mine Monitoring with Wireless Sensor Networks	Coal mine monitoring	Self-adaptive coal mine WSN system for rapid detection of structure variations caused by underground collapses
35	ITS [102]	2009	Wireless Sensor Networks for Intelligent Transportation Systems	Vehicle tracking and traffic monitoring	Traffic monitoring system implemented through WSN technology within SAFESPOT Project
36	Underwater [103]	2010	Adaptive Decentralized Control of Underwater Sensor Networks for Modeling Underwater Phenomena	Underwater networks	Measurement of dynamics of underwater bodies and their impact in the global environment, using sensor networks with nodes adapting their depth dynamically
37	PipeProbe [104]	2010	PipeProbe: A Mobile Sensor Droplet for Mapping Hidden Pipeline	Power line and water pipe monitoring	Mobile sensor system for determining the spatial topology of hidden water pipelines behind walls
38	Badgers [105]	2010	Evolution and Sustainability of a Wildlife Monitoring Sensor Network	Wild animal monitoring	Badger monitoring in a forest
39	Helens volcano [106]	2011	Real-World Sensor Network for Long-Term Volcano Monitoring: Design and Findings	Volcano monitoring	Robust and fault-tolerant WSN for active volcano monitoring

...

Table 3.1 – continued

Nr	Codename	Year	Title	Class	Description
40	Tunnels [107]	2011	Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels	Tunnel monitoring	Closed loop wireless sensor and actuator system for adaptive lighting control in operational tunnels

3.2.1 Deployment state and attributes

Full details of deployment state and used mote characteristics can be found in Appendix, see Table A.1.

Deployment state represents maturity of the application: whether it is just a prototype or a pilot test-run in real environment, or it has been running in stable state for a while. Only a few deployments are in stable state, the majority are prototypes and pilot studies. Therefore it is important to support fast prototyping and effective debugging mechanisms for these phases.

Despite theoretical assumptions about huge networks consisting of thousands of nodes only a few deployments contain more than 100 nodes. 80% of listed deployments contain 50 or less nodes, 34% - less than 10 nodes (Figure 3.1). Therefore communication stack included in the default operating system (OS) libraries, should concentrate on usability, simplicity and resource efficiency, rather than provide complex and resource intensive, scalable protocols for thousands of nodes.

Remote reprogramming is essential though, as it is very time intensive and difficult to program more than 5 nodes. And often nodes need many reprogramming iterations after initial setup at the deployment site.

The majority of deployments are built of homogenous networks with equal nodes: 70% of deployments. However, significant amount of deployments contain heterogenous nodes, and that must be taken into account in remote reprogramming design - users must be able to select subsets of network nodes to reprogram.

Almost all networks have a sink node or base station, collecting the data. Therefore sink-oriented protocols must be provided. Significant part of deployments use multiple sinks, which must be supported in the protocols.

Almost half of deployments use regular mote connected to a personal computer (usually a laptop) as a base station hardware solution. OS toolset must therefore include

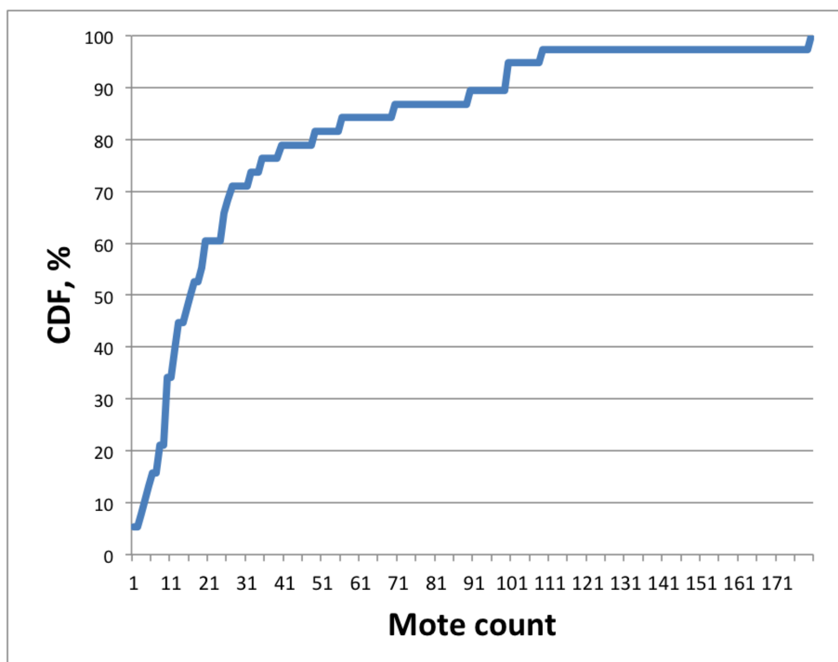

Figure 3.1: Distribution function of mote count in surveyed deployments - 80% of deployments contain less than 50 motes, 50% - less than 20 motes, 34% - ten or less

a default solution for base station application, which is easily extensible to user specific needs.

3.2.2 Sensing

Detailed description of sensing subsystem and sampling characteristics can be found in Appendix, see Table A.2.

The most popular sensors are temperature light sensors and accelerometers (Figure 3.2). Therefore WSN operating system should include API for temperature these in the default library set.

Interfaces used for sensor attachment, user feedback and interaction are listed in Table A.7. ADC is the most popular option for sensor interfacing: used in more than 50% of analyzed deployments, due to fact that majority of used sensors are analog.

Sensing applications may have two types of sampling: periodic, using timers, and event based - where data processing is triggered by sensed events. Both these approaches are used in sensor networks. Periodic sensing is more popular - 60% of applications use periodic sensor sampling and data processing, while pure event based approach is used in 22% of deployments. Part of deployments (18%) are hybrids - both, event triggered and periodic sensing is performed.

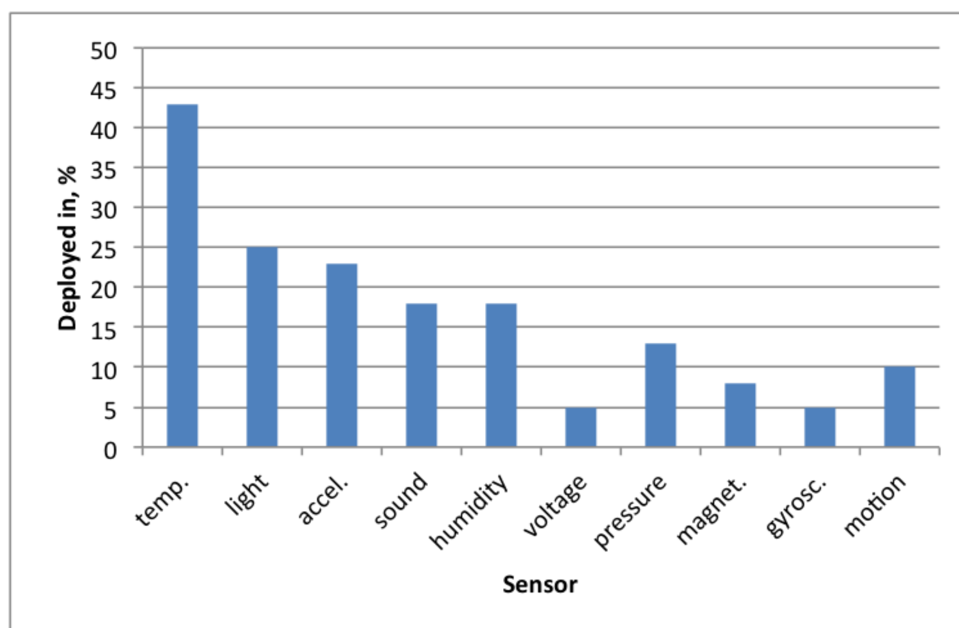

Figure 3.2: Sensors used in deployments - temperature, light and acceleration sensors are the most popular: each of them used in more than 20% of analyzed deployments

When considering sensor sampling rate, a pattern can be observed (Figure 3.3). Most of the deployments are low sampling rate examples, where the mote has a very low duty cycle, and sampling rate is less than 1Hz. Other, less popular application classes use sampling in the range 10-100Hz and 100-1000KHz. The former class uses accelerometer data processing while the latter is mainly representative of audio and high sensitivity vibration processing. Significant part of applications have variable sampling rate, configurable in run time.

Global positioning system (GPS) localization is widely used technology globally. However, it is not very popular in sensor networks, mainly due to unreasonably high power consumption. It is used in less than 18% of deployments. Therefore, GPS module should not be considered as a default component.

3.2.3 Lifetime and energy

Table A.3 (in Appendix) describes energy usage and target lifetime of analyzed deployments.

Target lifetime is very dynamic among applications: from several hours to several years. Long-living deployments use duty-cycle below 1%, meaning, that sleep mode is used 99% of the time. Therefore operating systems should provide effective routines for duty-cycling and have low computational overhead. Significant part of deployments (more

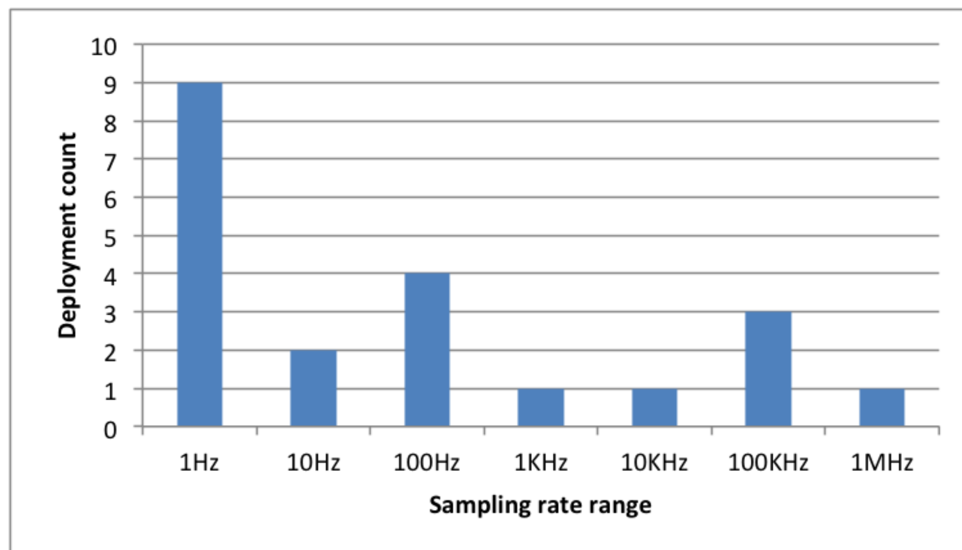

Figure 3.3: Sensor sampling rate used in deployments - Low duty cycle applications with sampling rate below 1Hz are the most popular, however, high-frequency sampling is also used, ranges 10-100Hz and 10-100KHz are popular

than 30%), especially in the prototyping phase, do not concentrate on energy efficiency and use 100% duty cycle. Automatic activation of sleep mode whenever possible would decrease the complexity and increase lifetime for deployments in prototyping phase, and also help beginner sensor network programmers.

Although energy harvesting is envisioned as the only way for sustainable sensing systems [108], power sources other than batteries or static power network are rarely used - in 5% of analyzed deployments. Harvesting module support in operating system level is, therefore, not essential part of deployments until today. However, harvesting popularity may increase in future deployments and their support could be a valuable option for WSN OS.

Both, very short and very long sleeping periods are used: from 250 milliseconds up to 24 hours. For convenient programming both, accurate short period sleeping and energy-efficient long period sleep modes should be supported by operating system.

More than 80% of deployments have power-motes present in the network: at least one node has increased energy budget. Usually, these motes are capable of running at 100% duty cycle, without sleep mode activation. This fact must be taken into account, when designing default networking protocol library.

3.2.4 Sensor motes

Table A.4 (in Appendix) lists used motes, radio (or other communication media) chips and protocols.

Mica2 [109] and MicaZ [110]platforms were very popular in early deployments. TelosB-compatible platforms (TMote Sky and others [32, 111, 112, 113]) are popular in recent years. Therefore TelosB platform support is essential for WSN OS.

Almost half of deployments (48%) use adapted versions of off-the-shelf motes by adding customized sensors, actuators and packaging. Almost one third (32%) use fully custom-built motes, by combining different microchips. Often these platforms are either compatible or similar to commercial platforms (for example, TelosB): the same MCUs and radio chips are used. Only 20% of deployments use off-the-shelf motes with default sensor modules. Therefore it is important for an operating system to support rapid:

- Implementation of additional sensor drivers for existing commercial motes.
- Porting to completely new platforms, providing effective mechanisms for existing commercial platform driver reuse.

The most popular reason for building a customized mote is specific sensing and packaging constraints. The application range is very wide and the author believes, that there will always remain need for customized platforms. Software support for customized platforms is therefore important.

On the other hand, part of sensor network users are beginners in the field and do not have resources to develop a new platform to assess a certain idea in real world settings. Off-the-shelf commercial platforms, simple programming interface, default settings and demo applications are required for this user class.

Chipcon CC1000 radio [114] was popular for early deployments, however, Chipcon CC2420 [8] is the most popular in recent years. IEEE 802.15.4 [34] is the most popular radio transmission protocol (used in CC2420 and other radio chips) at the moment, and with high confidence, it will keep the positions in near future. Driver support for CC2420 is essential. More radio chips and System-On-Chip (SoC) solutions using IEEE 802.15.4 protocol can be expected in the coming years.

3.2.5 Sensor mote: microcontroller

Used microcontrollers are listed in Table A.5, in Appendix.

Only a few deployments use motes with more than one MCU. Therefore the potential usage of OS support for multi-MCU platforms is limited. Multi-MCU motes is a

future research area for applications running simple tasks routinely and requiring extra processing power sporadically.

The most popular MCUs belong to Atmel ATMega and Texas Instruments MSP430 families. The former is used in Mica-family motes while the latter is the core of TelosB platform, widely used recently. Therefore support for these MCUs is essential for sensor network operating systems. A few ARM family processors are used, not very widely.

Sensor network motes use 8-bit or 16-bit architectures, with a few 32-bit ARM-family exceptions. Typical CPU frequencies are around 8MHz, RAM amount: 4-10KB, Program memory: 48-128KB. It must be noted, that program memory size is always larger than RAM, sometimes even by a factor of 32. Therefore RAM memory effective usage is more important and reasonable amount of program memory can be sacrificed for that matter.

3.2.6 Sensor mote: external memory

Used external memory characteristics are described in Table A.6 (see Appendix). While external memory of several megabits is available on most sensor motes, it is actually seldom used: only in 25% of deployments. Although very popular, Secure Digital/MultiMediaCard (SD/MMC) memory cards are even less frequently used: in less than 10% of deployments. Despite the fact that multiple sensor network file systems have been proposed previously [49, 115], they are not used. And, probably, there is a connection between (lack of) external memory and file system usage - external memories are rarely used, because there is no simple and efficient file system for these devices. Convenient file system interface should be provided by operating system, so that sensor network users can use it without extra complexity.

3.2.7 Communication

Table A.8 (in Appendix) lists deployment communication characteristics.

Data report rate varies a lot - some applications report once a day while others perform real-time reporting at 100Hz. If we search for connection between Table A.2 and Table A.8, two conclusions can be drawn:

- Low report rate is associated with low duty cycle.
- Low report rate not necessary implies low sampling rate - high-frequency sampling applications with low report rate do exist [78, 99, 100].

Typical data payload size is in the range 10-30 Bytes. However, larger packets are used in some deployments. Therefore default packet size provided by operating system should be around 30 bytes with option to change this constant easily, when required.

Typical radio transmission ranges are in the order of few hundred meters. Some deployments use long-range links with more than 1km connectivity range. For any deployment, option to change radio transmission power (if provided by radio chip) is a valuable option for collision avoidance and energy efficiency.

Data transmission speed is usually below 1MBit theoretically and even lower practically. It must be taken into account when designing communication protocol stack.

80% of deployments consider the network to be connected without interruptions - any node can communicate to other nodes at any time (not counting delays imposed by MAC protocols). Only 12% experience interruptions, and 8% of networks have only opportunistic connectivity.

Used communication media characteristics are listed in Appendix, Table A.9. With few exceptions, the communication is performed by transmitting radio signals over air. Ultrasound is used as alternative. And parts of networks may use available wired infrastructure.

85% of applications use one, static radio channel, the remaining 15% do switch between multiple alternative channels. While directionality usage for extended coverage and energy efficiency has been a widely discussed topic, the ideas are seldom used in practice. Only 10% of deployments use radio directionality benefits, and none of these deployments utilize electronically switchable antennas capable of adjusting directionality in real time [11].

3.2.8 Network

Deployment networking parameters are summarized in Table A.10 (see Appendix).

Mesh, multi-hop network is the most popular network topology - used in 47% of analyzed cases The 2nd most popular topology is simple one-hop network: 25%. Multiple such one-hop networks are used in 15% of deployments.

A surprising finding: almost half of deployments (47%) have at least one mobile node in the network. Therefore topology changes must be expected and handled correctly by routing protocol. Neighbor discovery protocols could also be required. Additionally: 30% have random initial node deployment, increasing the need for neighbor discovery protocol.

Maximum hop count does not exceed 11 in the surveyed deployments. Therefore, routing protocol scalability and routing table size are not critical.

3.2.9 Networking stack

Networking protocol stack is summarized in Table A.11, see Appendix.

43% of deployments use custom MAC protocols, proving that data link layer problems are very application-specific and important to assure energy-efficiency. Most commonly used MAC protocols can be divide into two classes: Carrier Sense Multiple Access (CSMA) based and Time Division Multiple Access (TDMA) based. The former class represents protocols which check media availability short before transmission while in the latter case all communication participants agree on a common transmission schedule.

70% use CSMA-based MAC protocols, 15% - TDMA and the remaining 15% is unclear. Conclusion: operating system should provide simple, effective and generic CSMA-based MAC protocol as default. TDMA MAC option would be a nice feature for WSN OS, as TDMA protocols are more effective in many cases. However, CSMA MACs are often used just because TDMA implementation is too complex: it requires master node election and time synchronization.

Routing is used in 65% of applications. However, no single best routing protocol is selected - between the analyzed deployment, no two applications used the same routing protocol. 43% of deployments used custom routing, not published before.

Routing can be proactive: routing tables are prepared and maintained beforehand; or it can be reactive - routing table is constructed only upon need. Proactive approach is used in 85% of the cases, the remaining 15% use reactive route discovery.

Therefore the conclusion: operating system must provide simple yet efficient, proactive routing protocol which performs fair enough for most of the cases. Interface for custom MAC and routing protocol substitution must be provided.

Although Internet Protocol version 6 (IPv6) is a widely discussed protocol and modifications (such as 6lowpan [116]) for resource constrained devices have been developed, the protocol is very novel and not widely used yet: only 5% of surveyed deployments use it. However, it can be expected, that this number will increase in coming years and most leading operating systems will adapt it. Sensor networks are essential part of the Internet-of-Things (IoT) movement that is accelerating recently. The IoT approach requires common standards with large address space. IPv6 is a promising standard for this matter.

Safe data delivery is used by 43% of deployments, showing that reliable communication in transport layer is a significant requirement for some application classes. Another quality-of-service option, data stream prioritizing, is rarely used though: only in 10% of cases. Conclusion: simple transport layer delivery acknowledgement mechanisms should be provided by the operating system.

3.2.10 Operating system and middleware

Used operating systems and middleware are listed in Table A.12 (see Appendix).

TinyOS [48] is de-facto operating system for wireless sensor networks: 60% of deployments use it. There are multiple reasons behind that. First, TinyOS has large community supporting it, therefore device drivers and protocols are well tested. Second, as it has reached critical mass, TinyOS is the first choice for new sensor network designers - it is being taught at universities, it has easy installation and well developed documentation, even books on how-to program in TinyOS [117].

At the same time, many C and Unix programmers would like to use their previous skills and knowledge to program sensor networks without learning TinyOS specifics, including nesC language (used by TinyOS) and component wiring. One evidence of this statement - new operating systems for sensor network programming are developed [16, 47, 49, 118, 119, 120], despite the fact, that TinyOS has been there for more than 10 years. Another evidence:in 25% of cases a self-made or customized OS is used: users either want to use their particular knowledge, or they have specific hardware, not supported by TinyOS and consider porting TinyOS to new hardware to be too complex.

Deluge [121] and TeenyLIME [64] middleware are used in more than one deployment. Deluge is a remote reprogramming add-on for TinyOS. TeenyLIME is middleware providing different level of abstraction, also implemented on top of TinyOS. Conclusion: middleware usage is not very popular in sensor networks. Therefore there is open space for research - to develop an easy to use yet powerful middleware, that is generic enough to be used in wide application range.

3.2.11 Software level tasks

User and kernel level tasks and services are described in Appendix, Table A.13. Task count and objectives are an estimate of the author of this deployment survey, developed based on information available from research articles. Networking, time synchronization and remote reprogramming protocols are considered kernel services, if not stated otherwise.

Most of deployments use not more than 2 kernel services: 55% (Figure 3.4). For some deployments up to 5 kernel services are used. Maximum service count must be taken into account when designing task scheduler.

In application layer, often just one task is used, which is typically sense and send: 33% of cases (Figure 3.5). Up to 6 tasks are used in more complex applications. Therefore task scheduler must support more than two application layer (user-space) tasks, preferably at least 5.

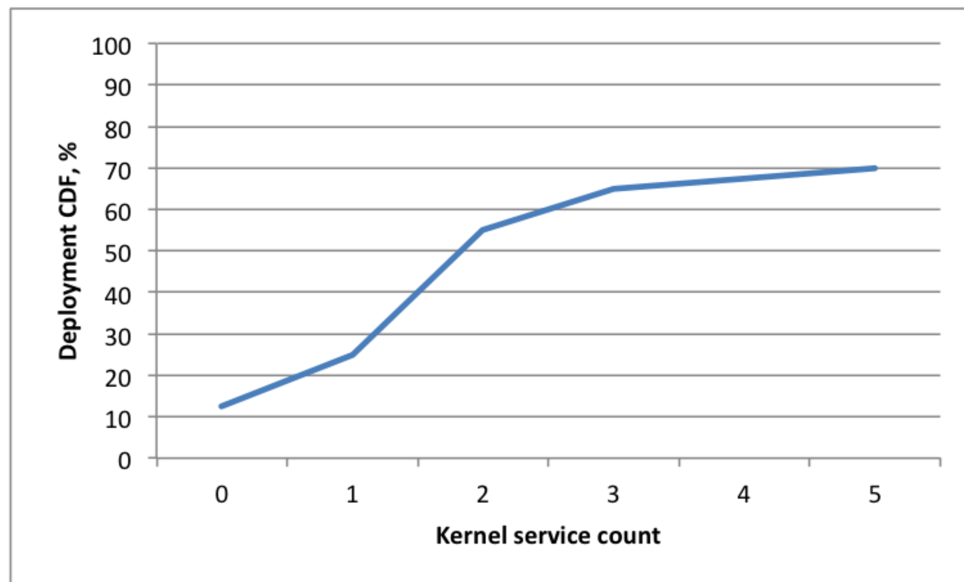

Figure 3.4: Number of kernel level software services used in deployments - 55% of deployments use 2 or less kernel services. For 28% kernel service count is unknown.

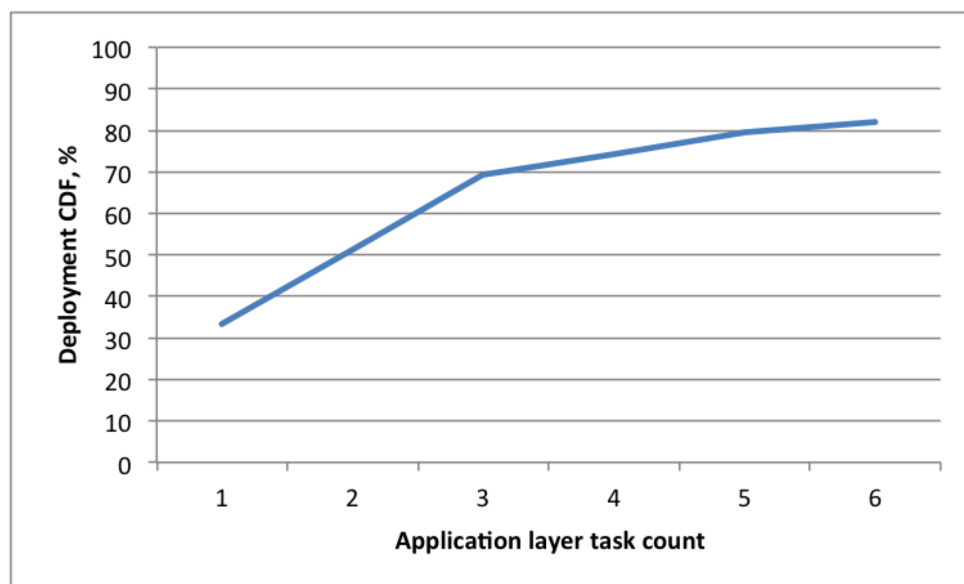

Figure 3.5: Number of application layer software tasks used in deployments - 33% of deployments use just one task, however, up to 6 tasks used in more complex cases. Task count is unknown in 18% of deployments.

3.2.12 Task scheduling

Table A.14 (in Appendix) describes deployment task scheduling attributes: time sensitivity and need for preemptive task scheduling.

Two basic scheduling approaches do exist: cooperative and preemptive. In the former case switch between tasks is explicit - one task yields processor to another task. Switch can occur only in predefined code lines. In the latter case, the scheduler can preempt any task at any time and give the CPU to another task. Switch can occur anywhere in the code.

The main advantage of cooperative scheduling is resource efficiency: no CPU time and memory is wasted to perform periodic switches between concurrent tasks, which could be executed serially without any problem. The main advantage of preemptive scheduling: users do not have to worry about task switching - it is performed automatically. Even if the user has created an infinite loop in one task, other tasks will have access to CPU and will be able to execute. However, preemptive scheduling can introduce new bugs: it requires context switching, including multiple stack management. Memory checking and overflow control is much harder for multiple stacks, compared to cooperative approaches with single stack.

If we assume, that user written code is correct, preemptive scheduling is required in a case, where at least one task is time sensitive and at least one other task is time intensive (can execute relatively long period of time). 20% of analyzed deployments have at least one time-sensitive application layer task (most of them: exactly one), while 30% of deployments require preemptive scheduling. Even in some cases (10%), where no user-space time-sensitive tasks do exist, preemption may be required by kernel-level services: MAC protocols and time synchronization.

Conclusion: operating system should provide both: cooperative and preemptive scheduling, switchable as needed. When using time-sensitive kernel services, scheduling should automatically switch to preemptive mode.

3.2.13 Time synchronization

Time synchronization has been addressed as one of the core challenges of sensor networks. Therefore its use in deployments is analyzed, statistics are shown in Table A.15 (see Appendix). Reliable routing is possible if at least one of two requirements holds:

a) 100% duty cycle is used on all network nodes functioning as data routers without switching to sleep mode.

b) Network nodes agree on a cooperative schedule for packet forwarding - time synchronization is required.

Therefore no effective duty cycling and multi-hop routing is possible without time synchronization.

Time synchronization is used in 38% of deployments, while multi-hop routing is used in 57% of cases (the remaining 19% use no duty-cycling).

Although very accurate time synchronization protocols do exist [122], simple methods, including GPS, are used most of the time, offering accuracy in millisecond, not microsecond range.

Only one of deployments used a previously developed time synchronization approach (not including GPS usage in two other deployments), all the others use custom methods. Reason: despite many published theoretical protocols, no operating system provides an automated and easy way to "switch on" time synchronization. Conclusion: time synchronization provided by the operating system would be of a high value, saving sensor network designers time and effort for custom synchronization development.

3.2.14 Localization

Another of the most addressed sensor network problems is localization, Table A.16 in Appendix.

Localization is used in 38% of deployments: 8% use GPS, 30% - other methods. In contrast to time synchronization, localization problem is very application specific. Required localization granularity, environment, meta-information and infrastructure varies tremendously: in one case, localization of centimeter scale must be achieved, in second: room of moving object must be found, in the third: GPS is used in outdoor environment. 73% of the cases, where localization is used, it is custom for this application. Therefore it is not possible for an operating system to provide a generic localization method for wide application class. Neighbor discovery service (as a part of multi-hop routing) could be usable.

3.2.15 Real-time data access

Real-time data access includes remote sensor reading access, software debug and remote reprogramming. Summary is shown in Table A.17 (see Appendix).

Remote data access and debug is used in 38%, remote reprogramming: 35%. It is clear, that remote, real-time data access is essential for efficient prototype and pilot installation where motes are in a specific packaging, or too many to plug each in a USB

port; or even worse - they are deployed on a hardly accessible remote site (such as island or volcano). Therefore remote debug, data access and reprogramming is an integral part of wireless sensor network operating system.

3.2.16 Discussion of future trends

In addition to deployment survey, the author has performed survey of recently designed hardware platforms for wireless sensor networks. Detailed results are summarized in Appendix, see Table B.1, Table B.2, and Table B.3. One conclusion from this platform survey - more powerful microcontrollers with ARM architecture emerge. However, the CPU frequencies are still less than 100MHz and amount of memory less than 100KiB. These still are mote-class low-power devices.

At the same time, Linux-based embedded platforms have evolved rapidly, including Raspberry Pi [56] and Odroid [57]. However, the energy consumption of these devices limit their application domain to different problems, requiring more complex real-time operations, such as image and audio signal processing.

Visual WSNs use high-performance platforms with Linux due to real-time large data sampling, storage and processing [123]. Energy consumption is in the order of watts for these platforms. If only image capture is required, devices with lower performance are useful (Cyclops [124]). High performance is required for object recognition and tracking.

For underwater networks the nodes can be significantly larger in size and weight, thus the main challenge is communication, not energy [125]. Linux can be used on these platforms.

In medical applications more trustworthiness and privacy must be present for large-scale, unsupervised deployments. Platform performance is not the limiting factor [126].

Sensor networks are mature enough to have more standardized protocols employed. The WSN field is contributing to growth of Internet-of-Things, where standardization is very important [127]. IEEE 802.15.4 and IPv6 over Low power Wireless Personal Area Networks (6loWPAN) are most widely used protocols [128]. In specific WSN domains other protocols might be dominating. For example, vehicular sensor networks are most likely to use Long Term Evolution (LTE) cellular technologies and IEEE 802.11n [129]. Although IPv6 and 6lowpan implementation for sensor networks is not popular at the moment, the popularity will increase at the moment, when internet hosts will start to actively use IPv6. One way of this shift is to start with mobile phones - introduce IPv6 in cellular networks.

Up to now sensor network applications have been developed mostly by networking researchers. As the field evolves and software abstractions provide more flexible access

for users without expertise in embedded systems, the count of non-expert applications could increase. There is a two-way link between attraction of new sensor-net designers and operating system improvement. Such users will, most probably, focus on application layer and use default networking protocols without developing custom implementations. New middleware, operating system add-ons or configurable application solutions can be expected, which provide sensor system access to people with no programming languages at all.

Energy harvesting methods are underestimated. The rise of harvesting technique usage can be expected, including harvesting from electro-magnetic field (EMF) energy, as cellular, WiFi and other wireless communication technology usage is growing and the EMF pollution grows accordingly.

People-centric sensing is rising in popularity due to high smartphone penetration speed. To extend wireless sensor networks to the next level, sensor mote cooperation with user-centric devices is essential. It can be accomplished by bridge devices having 802.15.4 and WiFi, BlueTooth or third generation of mobile telecommunications technology (3G) communication.

Radio-Frequency IDentification (RFID) and Near-Field Communication (NFC) technologies could grow in popularity. Especially if technology giants, such as Nokia and Google, support them [130, 131]. These technologies provide new application lifecycle model: wake-on-radio + process + respond, which could require new software paradigms.

New sensors, providing revolutionary applications, can be discovered. Although it is impossible to predict, what kind of sensors they will be, operating system must be flexible enough to support wide range of sensor connections, to allow rapid experimentation. New applications with ultra-low power consumption requirement could arise. However, most likely these systems will be entirely programmed by the designers, without usage of operating systems or middleware. Multiple serious environmental catastrophes have occurred during last years, suggesting, that environmental hazard detection and warning, as well as human rescue assistance systems could be developed in near future.

Intelligent Transportation Systems (ITS) is a promising approach to use information technologies for increased traffic safety, efficiency and comfort. Wireless communication is one part of ITS, and recently IEEE 802.11p standard has evolved to stable state [132]. Similarly to people-centric sensing, sensor motes can be used as eyes and ears of the environment to provide information for traffic participants. Bridging of 802.15.4 and 802.11p is required to create collaboration between mote world and ITS, or energy efficient 802.11p solutions must be designed.

One of the most popular platforms in the last years has been TMote Sky manufactured MoteIV (company renamed to Sentilla later). TMote mote is not manufactured

already since 2006, however, it is still used in academical research. In long term, other motes must take TMote's place and its established TelosB platform standard. Probably, new motes will replace components found on TelosB and incrementally shift away from the almost standard-like TelosB platform.

Traditional mote MCUs do not have memory protection mechanisms, while some powerful MCUs (for example, ARM family MCUs used on mobile phones) on the market have it. Sensor network systems could start adapting these devices with memory protection features more actively, as it would provide better debugging and application safety.

European global positioning satellite system Galileo was planned to launch in 2014, now postponed to 2019 [133]. This new technology would provide more accurate localization techniques and research of Galileo usage in sensor networks can be expected.

3.2.17 Summary

This is an important section describing results of a thorough analysis of a set of wireless sensor network deployments. The results have high impact on design rule development in Section 4. In addition, the analysis reveals, that many deployments use either customized or fully custom-built hardware platforms substantiating the hypothesis that WSN operating system must be portable to new hardware platforms. As the analysis shows, higher-performance microcontrollers are emerging, yet the mote-class sensor nodes are still present and will have also applications in future.

4 Sensor network software design rules

This chapter consists of three parts. First, typical WSN problems are identified. Second, WSN software development design rules are proposed. Third, examination is made on how the proposed design rules are addressing the identified problems.

Although every WSN application is unique and no rule can be defined as a theorem, these rules represent a unified methodology of WSN software development based on the best practice. By following the rules operating system and application developers ensure that the solution will meet most common efficiency, flexibility and reliability requirements of wireless sensor networks. The proposed design rules, with a focus on OS development aspect, have been published in a scientific journal article [30].

4.1 Problem identification

In this section the author identifies typical problems that arise in wireless sensor network software development due to WSN specific characteristics. The substantiation is based on analysis of existing WSN deployments and the author's experience in the sensor network research projects. Recognition of these problems is an important step in the beginning of WSN solution design. Solutions for the listed problems will be discussed in Section 4.3.

4.1.1 Portability and usability

The main problems related to portability and usability are the following:

Problem 1: **Chip reuse**. Various hardware platforms do exist that have common microchip and sensor base but different wiring and combination.

Problem 2: **Field experts**. WSN is a promising field not only for programmers and electrical engineers but also field experts with limited programming skills.

Problem 3: **Hardware evolution**. WSN users may choose different hardware platforms during the evolution of sensor network. However, the application logic and source code should be portable with minimal modifications.

4.1.2 Wireless communication

Problems related to wireless communication:

Problem 4: **Protocol variety**. Many WSN communication protocols do exist, yet not many ready-to-use implementations are available.

Problem 5: **WSN $\neq$ Internet**. WSN architecture is completely different from the Internet. Traditional protocols are not optimal, custom approach is required.

Problem 6: **Complex protocols**. Communication protocols are often too complex to provide full flexibility in unreliable networks.

Problem 7: **Limited resources**. WSNs must be able to communicate in dynamic topologies. However, memory and other resources are limited.

Problem 8: **Experimentation**. WSN researchers investigate and analyze protocols. An environment and infrastructure for experimentation is required.

Problem 9: **QoS**. A certain degree of Quality-of-Service (QoS) is required, especially in applications where WSNs are replacing traditional wired solutions.

4.1.3 Services and efficiency

Problems related to services and energy efficiency:

Problem 10: **Energy**. The central problem of WSNs is energy efficiency, yet many pilot and prototype deployments use 100% duty cycle. Such approach may incur significant loss of realism in these deployments.

Problem 11: **Data caching**. Dynamic Networks with probabilistic communication may require data caching and preprocessing. In addition, local data logging for redundancy might be important, especially during prototyping phases.

Problem 12: **Complex states**. It is hard for programmers to think in event-driven approach that requires explicit management of system state and split-phase operation. Also it is hard to design programs with multiple concurrent events within a single thread of execution.

Problem 13: **Cooperation**. The whole network of nodes should cooperate to reach the real benefits of WSNs: higher resolution, energy efficiency and cooperative decision making.

4.2 Design rule definition

In this section the author proposes wireless sensor network design rules that are based on existing WSN deployment analysis (described in Chapter 3). The importance of rules is divided into the following classes based on their popularity among analyzed deployments:

1. MUST: the feature must be implemented. Deployment support: 40-100%.
2. SHOULD: feature implementation has lower impact. Deployment support: 20-50%.

Feature classes are overlapping in terms of popularity in deployments, as it is hard to define sharp thresholds for feature popularity and strictly assign importance classes. Overlapping boundaries open space for discussion.

Proposed rules are divided into multiple categories based on addressed aspects of WSN development. Each of the following subsections lists proposed rules in one category. Table 4.1 lists all proposed rules in a summarized form.

Table 4.1: WSN software design rules proposed by the author

#	Rule	Description	Importance
		Communication	
1	Sink-oriented	The provided communication protocols must be sink-oriented.	MUST
2	Powered motes	Powered mote availability must be considered when designing a default networking protocol library.	MUST
3	30 byte payload	Default packet size should be at least 30 bytes with option to change this constant easily, when required.	SHOULD
4	11 hop routing	Multi-hop routing must be provided as a default component, which can be turned off, if one-hop topology is used. Topology changes must be expected, at least 11 hops should be supported.	MUST
5	CSMA MAC	A simple and generic CSMA-based MAC protocol must be included in WSN solutions, preferable as part of OS libraries.	MUST
6	Custom protocol API	Interface for custom MAC and routing protocol development must be provided.	MUST
7	Packet acknowledgment	Simple transport layer delivery acknowledgment mechanisms must be provided by the operating system.	MUST
8	IPv6 support	IPv6 (6lowpan) networking stack should be included in the operating system libraries to increase interoperability.	SHOULD
		Portability	
9	TelosB support	TelosB-compatible platform should be supported by WSN operating systems.	SHOULD

...

Table 4.1 – continued

#	Rule	Description	Importance
10	Rapid driver development	WSN operating systems must support implementation of additional sensor and other module drivers	MUST
11	Rapid platform definition	Porting to completely new platforms must be simple enough and operating systems should contain highly portable code.	MUST
12	802.15.4 support	Driver support for CC2420 radio or other 802.15.4-compatible radio communication chips should be provided by WSN operating systems.	SHOULD
13	AVR and MSP430 support	WSN operating systems must support Atmel AVR and Texas Instruments MSP430 MCU architectures.	MUST
		Task scheduling	
14	Low duty-cycle	WSN operating systems must set effective low-energy, low duty-cycle sampling as the first priority. High performance for sophisticated audio or other signal processing is secondary.	MUST
15	5 kernel + 6 user tasks	OS task scheduler must support up to 5 kernel services and up to 6 user level tasks.	MUST
16	Cooperative scheduling	Operating systems must provide cooperative tasks scheduling.	MUST
17	Preemptive scheduling	Operating systems should provide preemptive scheduling.	SHOULD
18	Event-based scheduling	Operating systems should provide event-based scheduling as an option.	SHOULD
		Services	
19	External storage	Interface for user data storage in external memory should be provided by WSN operating systems.	SHOULD
20	File system	Convenient file system interface should be provided by operating systems.	SHOULD
21	Time synchronization	Simple time synchronization should be provided by WSN operating systems.	SHOULD
		User support	
22	Base station example	WSN OS toolset must include an example base station application, which is easily extensible to user specific needs.	MUST
23	Popular sensor API	WSN operating system should provide common interface for temperature, light and acceleration sensor reading.	SHOULD
24	ADC API	ADC sampling interface must be provided by WSN operating systems.	MUST
25	Remote access	Remote data access and reprogramming of sensor nodes should be provided either by operating systems or other software abstractions.	SHOULD

4.2.1 Communication

Almost all (95% of deployments) networks have a sink node or base station, collecting the data. Significant part of deployments use multiple sinks.

Design rule 1: **Sink-oriented**. The provided communication protocols must be sink-oriented.

This rule implies that the communication flow is directed to a single collection point. This fact can be used to make protocols more efficient in terms of latency and memory resource allocation.

More than 80% of deployments have powered motes present in the network: at least one node has increased energy budget. Usually, these motes are capable of running at 100% duty cycle, without sleep mode activation.

Design rule 2: **Powered motes**. Powered mote availability must be considered when designing a default networking protocol library.

Typical data payload size is in the range 10-30 Bytes. However, larger packets are used in some deployments.

Design rule 3: **30 byte payload**. Default packet size should be at least 30 bytes with option to change this constant easily, when required.

Multi-hop routing is used in 57% of cases. Maximum hop count does not exceed 11 in the surveyed deployments. Almost half of deployments (47%) have at least one mobile node in the network (while maintaining a connected network).

Design rule 4: **11 hop routing**. Multi-hop routing must be provided as a default component, which can be turned off, if one-hop topology is used. Topology changes must be expected, at least 11 hops should be supported.

70% of deployments use CSMA-based MAC protocols, 15% - TDMA (15% unknown).

Design rule 5: **CSMA MAC**. A simple and generic CSMA-based MAC protocol must be included in WSN solutions, preferable as part of OS libraries.

Custom MAC or routing protocols are used in 60% of deployments. Therefore it is important to provide interface for custom protocol development.

Design rule 6: **Custom protocol API**. Interface for custom MAC and routing protocol development must be provided.

Reliable data delivery is used by 43% of deployments, showing that reliable communication in transport layer is a significant requirement for some application classes.

Design rule 7: **Packet acknowledgement**. Simple transport layer delivery acknowledgment mechanisms must be provided by the operating system.

Although IPv6 is a widely discussed protocol for the Internet of things and modifications (such as 6lowpan [116]) for resource constrained devices have been developed, the protocol is very novel and not widely used yet: only 5% of surveyed deployments use it. However, it can be expected that this number will increase in coming years. TinyOS [48] and Contiki OS [47] have already included 6lowpan as one of the main networking alternatives.

Design rule 8: **IPv6 support**. IPv6 (6lowpan) networking stack should be included in the operating system libraries to increase interoperability.

4.2.2 Portability

TelosB-compatible platforms (TMote Sky and others [111, 112]) are the most popular among WSN hardware platforms in recent years.

Design rule 9: **TelosB support**. TelosB-compatible platform should be supported by WSN operating systems.

Almost half of deployments (47%) use adapted versions of off-the-shelf motes by adding customized sensors, actuators and packaging. Almost one third (32%) use custom motes, by combining different microchips. Often these platforms are either compatible or similar to commercial platforms (for example, TelosB): use the same microcontrollers (MCUs) and radio chips.

Design rule 10: **Rapid driver development**. WSN operating systems must support implementation of additional sensor and other module drivers.

Design rule 11: **Rapid platform definition**. Porting to completely new platforms must be simple enough and operating systems should contain highly portable code.

Chipcon CC2420 [8] radio chip is the most popular in recent years and IEEE 802.15.4 [34] was found to be the most popular radio transmission protocol (37.5% of deployments).

Design rule 12: **802.15.4 support**. Driver support for CC2420 radio or other 802.15.4-compatible radio communication chips should be provided by WSN operating systems.

The most popular MCUs belong to Atmel ATMega (AVR architecture) and Texas Instruments MSP430 families. The former is used in Mica-family motes while the latter is the core of TelosB platform, widely used recently. In total, 80% of deployments use either AVR or MSP430 based microcontrollers.

Design rule 13: **AVR and MSP430 support**. WSN operating systems must support Atmel AVR and Texas Instruments MSP430 MCU architectures.

4.2.3 Task scheduling

Energy-efficiency is the core requirement of sensor network longevity. Low-duty cycle operation with low-frequency sensor sampling is preferred by 48% of deployments.

Design rule 14: **Low duty-cycle**. WSN operating systems must set effective low-energy, low duty-cycle sampling as the first priority. High performance for sophisticated audio or other signal processing is secondary.

Most of deployments use not more than 2 kernel services: 55% (Figure 3.4 in Section 3.2.11). For some deployments up to 5 kernel services are used. In application layer, often just one task is used, which is typically sense and send: 33% of cases (Figure 3.5 in Section 3.2.11). Up to 6 tasks are used in more complex applications.

Design rule 15: **5 kernel + 6 user tasks**. OS task scheduler must support up to 5 kernel services and up to 6 user level tasks.

An alternative configuration might be useful, providing a single user task to simplify programming approach and provide maximum resource efficiency that might be important for the most resource constrained platforms.

30% of deployments require preemptive scheduling. For others, cooperative scheduling is preferred as more resource-efficient.

Design rule 16: **Cooperative scheduling**. Operating systems must provide cooperative tasks scheduling.

Design rule 17: **Preemptive scheduling**. Operating systems should provide preemptive scheduling.

The scheduling techniques should be switchable by configuration at the source code compilation stage.

Although event-based programming might be complex for programmers with conventional experience with desktop systems, embedded systems are event-driven by nature and such programming paradigm might be most effective in some application scenarios. In addition, event-driven programming is popular in smartphone operating systems, such as Android. Therefore, more programmers adapt to this paradigm.

Design rule 18: **Event-based scheduling**. Operating systems should provide event-based scheduling as an option.

4.2.4 Services

External memory is used in 27.5% of deployments, and filesystem is used only in 5% of deployments. This presents a different case, compared to other design rule inference. The author believes, that the reason why external memory is seldom used, is the fact that operating systems do not provide simple-to-use file system. Further studies should be performed to verify this hypothesis.

Design rule 19: **External storage**. Interface for user data storage in external memory should be provided by WSN operating systems.

Design rule 20: **File system**. Convenient file system interface should be provided by operating systems.

Time synchronization is used in 37.5% of deployments.

Design rule 21: **Time synchronization**. Simple time synchronization should be provided by WSN operating systems.

4.2.5 User support

Almost half of deployments use regular mote connected to a PC (usually a laptop) as base station hardware solution.

Design rule 22: **Base station example**. WSN OS toolset must include an example base station application, which is easily extensible to user specific needs.

The most popular sensors are temperature, light and accelerometer sensors, used in 42.5%, 25% and 22.5% respectively (Figure 3.2 in Section 3.2.2).

Design rule 23: **Popular sensor API**. WSN operating system should provide common interface for temperature, light and acceleration sensor reading.

50% of deployments use analog sensors that are sampled using analog-to-digital converter (ADC). Therefore ADC should be treated as a core module of each sensor node.

Design rule 24: **ADC API**. ADC sampling interface must be provided by WSN operating systems.

Remote node access is used in 38% of deployments. It is an essential part of testing a sensor network.

Design rule 25: **Remote access**. Remote data access and reprogramming of sensor nodes should be provided either by operating systems or other software abstractions.

4.3 Addressing problems by rules

This section analyzes relation between identified WSN problems and proposed design rules. For each problem it is discussed how and which of the proposed design rules can reduce the consequences. Results show that proposed design rules have an M:N relation to common WSN problems: each problem is addressed by multiple rules and each rule can be used to mitigate multiple problems. Summary of design rule correspondence to problems is shown in Table 4.2. Details will be discussed in the following subsections.

4.3.1 Portability and usability

Here we analyze how proposed design rules address problems related to portability and usability:

Problem 1: **Chip reuse**. This problem can be solved by providing drivers for popular modules (*design rules #9, #12, #13*), rapid new driver development (*design rules #10 and #24*) and support for reconfiguration of wiring for hardware platforms (*design rules #10 and #11*).

Problem 2: **Field experts**. The solution is to provide ready-to-use components for complex parts of WSN software and allow the users to focus on application. The complex parts include networking protocols according to *design rules #4, #5 and #8*; TelosB platform support (*design rule#9*); task schedulers (*design rules #15, #16, #17*); different services, interfaces and examples (*design rules #19 - #25*).

Table 4.2: Addressing WSN problems by design rules (rules in rows, problems in columns)

	Design rules	#1 Chip reuse	#2 Field experts	#3 Hardware evolution	#4 Protocol variety	#5 WSN $\neq$ Internet	#6 Complex protocols	#7 Limited resources	#8 Experimentation	#9 QoS	#10 Energy	#11 Data caching	#12 Complex states	#13 Cooperation
	Communication rules													
1	Sink-oriented					x	x	x						
2	Powered motes					x	x				x			
3	30 Byte payload					x								
4	11 hop routing		x		x	x		x			x			
5	CSMA MAC		x		x	x								
6	Custom protocol API								x	x	x			
7	Packet acknowledgment				x	x				x		x		
8	IPv6 support		x		x	x								
	Portability rules													
9	TelosB support	x	x											
10	Rapid driver dev.	x		x										
11	Rapid platform def.	x		x										
12	802.15.4 support	x												
13	AVR & MSP430 support	x												
	Task scheduling rules													
14	Low duty-cycle										x			
15	5 kernel + 6 user tasks		x										x	
16	Cooperative sched.		x								x		x	
17	Preemptive sched.		x										x	
18	Event-based sched.							x			x			
	Service rules													
19	External storage		x									x		
20	File system		x									x		
21	Time Synchronization		x								x			x
	User support rules													
22	Base-station example		x											
23	Popular sensor API		x	x										
24	ADC API	x	x	x										
25	Remote access		x						x					

Problem 3: **Hardware evolution**. Rapid driver development (*design rule#10*) and porting to new hardware platforms (*design rule#11*) is essential. In addition, unified interface for sensor sampling supports different platforms without changing application code (*design rules #23 and #24*).

4.3.2 Wireless communication

This section summarizes design rules addressing problems related to WSN communication:

Problem 4: **Protocol variety**. Implementation of basic, general purpose communication protocols should be provided according to *design rules #4, #5, #7, #8.*

Problem 5: **WSN $\neq$ Internet**. Provided protocols should be adapted for WSN context, according to *design rules #1-#5, #7, #8.*

Problem 6: **Complex protocols**. Specific aspects can be used to make protocols simpler: consider powered motes and sink-oriented nature of data flow (*design rules #1 and #2*).

Problem 7: **Limited resources**. Limited network size (*design rule#4*) and sink-orientation of data flow (*design rule#1*) can be exploited to create simpler routing algorithms requiring less computation and memory. Also, programming in event-driven paradigm requires less memory overhead (*design rule#18*).

Problem 8: **Experimentation**. Interface for development of custom communication protocols must be provided (*design rule#6*). Remote access is suggested for advanced debugging of systems in test environment (*design rule#25*).

Problem 9: **QoS**. Packet acknowledgements can be integrated into the transport layer of networking protocol stack according to *design rule#7.* Users may want to implement custom QoS requirements, therefore interface for custom protocol definition is important (*design rule#6*).

4.3.3 Services and efficiency

This section discusses how design rules can mitigate WSN problems related to services and energy efficiency:

Problem 10: **Energy**. Operating system and other existing WSN tools should provide framework for energy-efficient operation, including low-duty cycle task scheduling

(*design rules #14, #16 and #18*) and multi-hop routing (*design rule#4*) that organizes transmissions so that nodes can listen for incoming transmissions only at certain time periods. Powered motes (*design rule#2*) or time synchronization (*design rule#21*) can be used for efficient transmission schedule management . Interface for custom protocol design (*design rule#6*) is also important in this aspect - sensor network environment can be specific for each application, MAC and routing protocol modifications might be needed to create optimal solution.

Problem 11: **Data caching**. Local data storage in external memory (*design rules #19, #20*) should be provided and mechanism for checking which data packets have been delivered to the destination (*design rule#7*).

Problem 12: **Complex states**. Multitasking system with linear scheduling should be provided allowing to run multiple concurrent threads according to *design rules #15-#17.*

Problem 13: **Cooperation**. Time synchronization and network-wide duty cycle is required for cooperation according to *design rule#21.* Another option is to include powered motes (*design rule#2*) in the network that act as buffers and intermediate relays for nodes that have different activity periods.

4.4 Summary

In this section the author identified typical WSN problems and proposed design rules that address these problems. The proposed design rules build a methodology for WSN software development and represent the core contribution of this thesis.

Problems and design rules do not have 1:1 relation, rather M:N (many-to-many). Each design rule addresses multiple problems and each problem can be mitigated by following different rules. Design rule set is consistent and does not have contradictory rules.

The proposed methodology, in the form of design rule set, functions as guidelines for successful wireless sensor network software development. Conformance to rules must be ensured at two levels. First, if used operating system or middleware conforms to proposed rules, it makes WSN development easier for users and application developers. Second, actual application developed by the users can be checked against the rules. WSN solution can be optimal only if rules are followed at both these levels. If operating system does not conform to rules, it will result in more time and resources spend by the users during application development. If the users do not follow the rules (even though everything is

correct from the OS point of view), the resulting solution will not be optimal in terms of energy efficiency and portability.

The proposed design rules can also be used as a checklist for WSN solution assessment. By checking conformance to rules potential problems can be identified in each individual case.

The following chapter evaluates proposed design rule methodology and shows how it can be applied to evaluate, design and improve different WSN software solutions.

5 Design rule impact on existing systems

This chapter discusses how the proposed design rules are applicable to analyze existing WSN software. Different aspects of software are analyzed with the goal to show how the solutions could be improved by following the proposed design rules. Different abstraction levels of WSN software are analyzed: operating systems and user applications (deployments). Although analysis of middleware could be interesting, it is out of the scope of this thesis and can be considered part of future work.

5.1 Impact on deployments

During WSN deployment survey the author has identified common trends than are formed as a set of design rules for WSN software development. However, no rule is satisfied in 100% of analyzed deployments. Not all deployments are optimal. While it is clear that deployments are different in terms of environment, research goals and constraints, in some cases efficiency of WSN applications can be improved by adapting proposed design rules for particular deployments. This section analyzes how surveyed deployments could be improved by making them according to proposed design rules.

- **Rapid driver development and porting (design rules #10 and #11)**. 80% of deployments have involved custom driver development for either platform with specific components or porting the same application to another or completely custom platform (32.5%). Source code portability is important for WSNs as the platforms often evolve and development follows the prototyping model. Existing code modules as well as the operating system ideology and structure should support rapid and frequent changes. Unfortunately, the most popular operating system, TinyOS, follows ideology and contains source code that is hard to read and understand (distributed in various places, contains nesC specific constructs), and even harder to design during porting. While TinyOS might have high performance and resource efficiency, it

should be improved dramatically in terms of usability. Although TinyOS us used here as an example (most popular OS choice among deployments), the portability and driver development rules are important for any WSN OS as these aspects impact many deployments.

- **Sink-oriented protocols and powered motes (design rules #1 and #2)**. 38 of 40 deployments (95%) needed a sink-oriented protocol and in 11 cases (27.5%) it was not provided by the operating system. Providing such protocol at OS or library level saves development time for users. Development of communication protocols is a complex task requiring thorough testing either in simulations (which might not accurately describe real-world scenarios) or real pilot networks (which may not always be available and require time-consuming software update procedures). Similarly, powered motes should be considered in these protocols (used in 82.5% of deployments, not provided by OS in 22.5%). By providing such protocols at OS level or libraries, users of 22.5% of deployments could have improved software development speed.

- **Custom protocol interface (design rule #6)**. Interface for definition of custom MAC and routing protocols is essential part of operating system or middleware - this feature is required by 62.5% of deployments, and the requirement is satisfied only in 68% of cases when it is needed (42.5% of total deployments). This rule could decrease development time for WSN protocol researchers and encourage testing protocols on real platforms, as well as simulations, if the OS allows to compile application for simulated sensor nodes.

- **Cooperative scheduling (design rule #16)**. Preemptive scheduling is only needed in 30% of cases. In 62.5% cases a cooperative scheduling strategy is sufficient (7.5% of cases do not have enough information). That implies that cooperative scheduling should be preferred as it is more efficient in different aspects, including efficient memory usage, less context switch time overhead and more appropriate task switch time selection. The design rule that suggests cooperative scheduling could improve 15% of deployments where cooperative scheduling is not provided by an operating system.

- **Popular sensor API (design rules #23)**. In 65% of cases at least one of the most popular sensors (light, temperature, accelerometer) is used. Therefore operating system or middleware should provide a unified API for these sensor sampling. Some platform inspection functionality should be available telling the application what

sensors are available. Unfortunately, such API is provided only in one of the cases where specific sensor extension board is used. By doing so one can assure that the same application can be run on different platforms.

- **Time synchronization (design rule #21)**. In 35% of deployments some form of time synchronization is implemented in the application. Proper time synchronization requires complex algorithms, similarly to network protocols. Therefore it would be valuable to include basic time synchronization in the operating system. In 7.5% of deployments advanced and application specific time synchronization is used, which cannot be implemented at the OS level. However, in most of cases a generic time synchronization would suffice.

5.2 Impact on operating systems

This section summarizes existing WSN operating system conformance to proposed design rules and analyzes improvements by applying the rules. The author analyzed operating systems, described in Section 2.1, and results are summarized in Table 5.1. The following sections will describe results for each of the presented operating systems.

5.2.1 TinyOS

As TinyOS complies to majority of rules, only the non-satisfied rules will be discussed in detail.

TinyOS disregards the following design rules: *design rule#10* (rapid driver development), *design rule#11* (rapid platform definition), *design rule#20* (file system), *design rule#21* (time synchronization), *design rule#23* (popular sensor API), and *design rule#25* (remote access). Although TinyOS is portable (wide range of supported platforms is a proof for it), code readability and simplicity is doubtful. The main reasons of TinyOS complexity are:

- The event-driven nature: while event handlers impose less overhead compared to sequential programming with blocking calls and polling, it is more complex for programmers to design and keep in mind the state machine for split-phase operation of the application.

- Modular component architecture: high degree of modularity and code reuse leads to program logic distribution into many components. Each new functionality may require modification in multiple locations, requiring deep knowledge of internal system structure.

Table 5.1: Existing OS conformance to proposed design rules

#	Rule	TinyOS	Contiki	LiteOS	Mantis	MansOS	Arduino
	Communication						
1	Sink-oriented	+	+	+	+	+	
2	Powered motes	+	+			+	+
3	30 byte payload	+	+	+	+	+	+
4	11 hop routing	+	+	±	+	+	+
5	CSMA MAC	+	+			+	+
6	Custom protocol API	+	+			+	
7	Packet acknowledgment	+	+		+	+	+
8	IPv6 support	+	+				+
	Portability						
9	TelosB support	+	+		+	+	
10	Rapid driver development		+	+	+	+	+
11	Rapid platform definition		±		±	+	
12	802.15.4 support	+	+	+	+	+	+
13	AVR and MSP430 support	+	+	±	+	+	
	Task scheduling						
14	Low duty-cycle	+	+	+	+	+	
15	5 kernel + 6 user tasks	+	+	+	+	±	
16	Cooperative scheduling	+	+			+	
17	Preemptive scheduling	+	+	+	+	+	
18	Event-based scheduling	+	+			+	
	Services						
19	External storage	+	+	+	+	+	+
20	File system		+	+	+	+	+
21	Time synchronization		+			+	+
	User support						
22	Base station example	+		+	+	+	+
23	Popular sensor API			+	±	+	+
24	ADC API	+		+	+	+	+
25	Remote access		+	+	+	+	±

- nesC language peculiarities: confusion of interfaces and components, component composition and nesting, specific requirements for variable definitions are examples of language aspects interfering with creativity of novice WSN programmers.

These limitations are in the system design level, and there is no quick fix available. The most convenient alternative is to implement middleware on top of TinyOS for simplified access to non-expert WSN programmers. TinyOS architecture is too specific and complex to introduce groundbreaking improvements for readability while maintaining backwards compatibility for existing applications.

Nevertheless, more than 100 groups around the world use TinyOS. It is also used by multiple commercial products [134, 135].

The rest of unsatisfied design rules regard to missing features that can be implemented as additions. And some of the functions are already implemented as external tools and middleware on-top of TinyOS. For example, third party external storage filesystem implementations do exist, such as TinyOS FAT16 support for SD cards [136]; Deluge can be used for remote reprogramming [121].

5.2.2 Contiki

Contiki does not provide platform independent API for temperature, light, and sound sensors (*design rule#23* (popular sensor API)) and ADC access (*design rule#24* (ADC API)). The reason is Contiki's mission - it is not dedicated specifically to sensor networks, rather to networked embedded device programming. Some of the platforms (such as Apple II) may not have sensors or ADC available, therefore the API is not explicitly enforced for all the platforms.

Portability to new platforms is partially effective (*design rule#11* (rapid platform definition)). MCU architecture code may be reused. However, large proportion of platform-specific code in Contiki may actually be reused on multiple platforms with appropriate restructuring.

Surprisingly, there is no base station application template included (*design rule#22* (base station example)). Contiki-collect is provided as an alternative - a complete and configurable sense-and-send network toolset for simple setup of simple sensor network applications.

To conclude, Contiki is one of the best WSN operating systems conforming with most of the proposed design rules.

5.2.3 LiteOS

LiteOS provides fully threaded programming with blocking calls, and no event callback handling (*design rule#18* (event-based scheduling)). No cooperative scheduler is provided (*design rule#16* (Cooperative scheduling)).

Networking stack is not included in the LiteOS distribution. However, multiple demo applications are usable as templates for user-specific networking protocol creation. Several routing protocols are implemented as user-level threads. The following communication rules are not satisfied: *design rule#2* (powered motes), *design rule#5* (CSMA MAC), *design rule#6* (custom protocol API), *design rule#7* (packet acknowledgement) and *design rule#8* (IPv6 support). LiteOS' applicability is very limited due to these inconsistencies.

LiteOS conforms to all user support rules, including interface for temperature, light and acceleration sensor sampling(*design rule#23*) and remote access (*design rule#25*). From service rules it lacks time synchronization support (*design rule#21*).

The source code is 8-bit AVR platform specific and significant changes are required to port LiteOS to other platforms with other microcontrollers. Chip driver development is relatively simple, as device drivers must implement only a predefined set of functions. However, new platform specification is unclear (*design rule#11* is not satisfied).

Compared to other WSN operating systems (TinyOS, Contiki and MansOS) LiteOS is a constrained OS with limited usability for field experts and other programmers who do not want to study or develop customized networking protocols.

5.2.4 Mantis

A TDMA-class MAC protocol supporting star network topology is included in the default configuration, without a CSMA MAC (*design rule#5* is not satisfied). No unified networking API is used, therefore users must design inter-layer interfaces on demand (*design rule#6* is not satisfied). In addition, the following rules are not satisfied by the existing networking implementation: *design rule#2* (powered motes), *design rule#7* (packet acknowledgement) and *design rule#8* (IPv6 support). Networking protocol stack of Mantis is not thoroughly developed, and development of the OS itself has stopped in recent years.

Platform- and chip-level code is mixed, there are no TelosB or MicaZ platforms, rather MSP430 and AVR code, which is MCU or architecture specific. Separation of MCU architectures, specific chips and platforms would improve portability (*design rule#11*).

Only preemptive thread scheduling is supported by the OS which, similarly to LiteOS, limits its efficiency for constrained application class. No cooperative scheduling (*design rule#16*) or event-based scheduling (*design rule#18*) is provided.

Mantis is rich in supported API, services and examples, yet no time synchronization is provided (*design rule#21*).

Mantis has a promising software base that would be extensible for a rich WSN OS. Unfortunately, it's development activity has stopped.

5.2.5 MansOS

MansOS is a wireless sensor network operating system developed with simplicity of use and portability in mind. The author has also participated in MansOS development since it's start in 2008. MansOS is still actively developed now. More MansOS details in Section 2.1.1.

During it's thorough development, ideology and core components of MansOS have evolved in multiple iterations. Therefore the existing version conforms to most of the design rules proposed in this thesis. There are some exceptions and space for improvement that will be discussed here.

The author of this thesis introduced cooperative task scheduler (described in Appendix, Section C.1.2.1) to MansOS as a result of design rule development (*design rule#16*). Previously MansOS was supporting two scheduling techniques: direct event handling and preemptive scheduler. Based on the findings in deployment survey development team decided that cooperative scheduler is an important part of WSN OS. As a result of multiple alternative evaluation, the author decided to integrate *ProtoThreads* scheduler from Contiki OS [137] - it has been proved to work stable already in Contiki, therefore there was no need to *reinvent the wheel*. It is an example of how design rules improved WSN OS in practice - by substantiating importance of a particular feature that was not implemented previously.

Two improvements are required in MansOS to reach full conformance to proposed design rules. First, IPv6 support is required (*design rule#8*). While third-party IPv6 libraries can be used [138], such addition would interfere with the existing networking protocol infrastructure. IPv6 should be fully integrated as an optional component in the common protocol stack. Second, the preemptive scheduler is limited to only one kernel thread at the moment while *design rule#15* states that 5 kernel tasks should be supported. This can be fixed by implementing a multi-threaded kernel, although it might require some re-design of the whole OS. Otherwise, two problems may arise. First, the tasks running in kernel context have equal priority, it is not possible to assign higher

priority to any of the tasks. Second, the kernel tasks are implicit without possibility to create libraries of additional kernel tasks, that can be loaded and unloaded as necessary.

In summary, MansOS demonstrates how design rules are applied both during OS development phase and also in evaluation to detect potential problems and design necessary improvements.

5.2.6 Arduino

This section discusses how Arduino conforms to proposed WSN software development design rules and how Arduino can be modified to become a fully-functional WSN OS.

Core Arduino OS provides only basic MCU driver and a base for extensions. However, Arduino is a community-based project without strict borders of OS and third-party software. It is a set of solutions and libraries that are combined during custom solution development. Therefore here we examine the opportunities of core Arduino together with libraries and extensions that are widely accepted. As the Arduino solution is based on engineer and enthusiast community (instead of WSN researchers) most of the references in this section point to web sites instead of scientific articles. Nevertheless, these sites do not have hypothesis and statements that have to be proved. Instead they contain source code libraries and examples that can be simply verified empirically. Therefore these sources are sufficiently reliable for this particular section.

Arduino core contradicts to all network rules as there is only a microcontroller on the base board and USB is the only communication with a PC. However Zigbee/XBee-802.15.4 modules are available providing networking options. Zigbee has a built-in mesh capability. Networking rules are described here, using XBee Series 2 modules with Zigbee stack [139]. In addition, new versions of Arduino boards with built-in communication do appear, such as Arduino Yun [140] and Flutter [141].

The Zigbee networking protocol stack implements an 802.15.4-compatible (*design rule#12*) mesh network topology, without sink-oriented data flow architecture (*design rule#1* is not satisfied). Powered motes are considered in Zigbee, called *coordinator* [142] (*design rule#2* is satisfied (powered mote support)). Zigbee includes CMSA-based MAC protocol and Ad hoc On-demand Distance Vector (AODV) routing [143] with reliable packet delivery (*design rules #4, #5 and #7* are satisfied). IPv6 library from Contiki has been ported to Arduino [144] (*design rule#8* satisfied). However, the MAC and routing protocols are predefined and cannot be customized by the user (*design rule#6* not satisfied).

Arduino platform has space for improvement regarding portability. It is designed only for AVR-based microcontrollers, TelosB platform (*design rule#9*) and MSP430-

family MCUs (*design rule#13* (AVR and MSP430 support)) are not supported. Arduino is not designed for other architectures, therefore its software is not ready for porting: *design rule#11* (rapid platform definition) is not satisfied. Nevertheless, driver development (*design rule#10*) is facilitated with wide range of existing sensor and other chip drivers, libraries and examples.

Task scheduling in Arduino is not optimal for WSNs. It uses a simple single-thread polling approach. Protothread library (from Contiki) can be used on Arduino [137], but it there is no ready-to-use Protothread port or tutorial available. All task scheduling design rules are not satisfied. An advanced task scheduler is needed to adapt Arduino for WSN needs.

Arduino supports wide range of services and interfaces using community-contributed libraries, all service and user support design rules are satisfied. Examples include external memory and FAT file system [145], time synchronization [146], ADC and wide range of sensors drivers [147]. Remote reprogramming is possible by external tools requiring custom bootloader [148].

To summarize, Arduino needs addition of task scheduler, portability to low-power hardware platforms and more flexibility for networking protocols.

5.2.7 Summary

The evaluation shows that popular WSN operating systems conform to majority of proposed design rules. However, each OS has some specific aspects that can be improved. This thesis can serve as a reference for OS developers substantiating importance of particular design rules.

5.3 Use case study: wearable sensor network

This case study describes a research project on tactile ship bridge alarm system development, performed jointly by Maritime Human Factors Laboratory at Aalesund University College and Rolls Royce Marine, Norway. The author was (and still is) participating in the project as part of the research team and was one of the main contributors in software and hardware design and development. The author had built the first prototype of the system at the time of writing this section. In this case study the author analyzes possible improvements of the tactile alarm system that can be introduced by implementing the proposed design rules.

5.3.1 Research problem

Ship bridges are operated by complex systems that integrate many subsystems [149]. Dissemination of alarms is an essential function of the system. Alarms from all subsystems are gathered in the central system and displayed for persons in charge, mainly to operators active at the moment. If there is no reaction from the operators, alarms are forwarded to other facilities, including captains and crew living rooms [150].

Bridge Navigational Watch Alarm System (BNWAS) is a specific and well standardized part of ship bridge alarm systems [151]. It sounds an alarm whenever watch officer on the bridge falls asleep or otherwise becomes unresponsive for too long [152]. The presence and awareness of watch officers must be either confirmed by pressing specific buttons or automatically by motion detection sensors [153]. Audio alarms are defined as mandatory by different standards [151, 154] and they are treated as natural due to their scope of operation properly installed audio alarms can reach all the crew members in any position and orientation.

The purpose of this research project is to challenge the notion that alarms are equal to audio sensory input. It investigates the possibilities to extend alarms beyond audio cues. Nevertheless, it is clear that tactile alarms will not replace audible alarms, at least not in near future. Tactile must be treated as a complimentary sensory input, used in combination with existing audible and visual alarm systems. Previous research shows that tactile stimuli has shorter response time and might help operators to distinguish between different types of alarms [155].

5.3.2 Approach

The author has designed system architecture, shown in Figure 5.1. It receives alarms from the ship bridge system, including information about desired recipients and location of focus. The system consists of two parts. One part is implemented as an add-on for the ship bridge system. It takes information about person location and actual alarms, and generates tactile alarm signals to be sent to persons. The second part is a system worn on the operators. It receives commands wirelessly and generates tactile cue patterns for the actuators mounted on the person.

There are several options for device types to be used. The author chose tactile belt as the most appropriate. Ideally it would be a smart belt that men wear as usual during the stay onboard. But in first iterations it will be a stretchable add-on type belt. It can be worn over the regular belt or adjusted around the abdominal. Its main advantages: close contact, naturalness (immersiveness) that leads to low human resistance, ability to

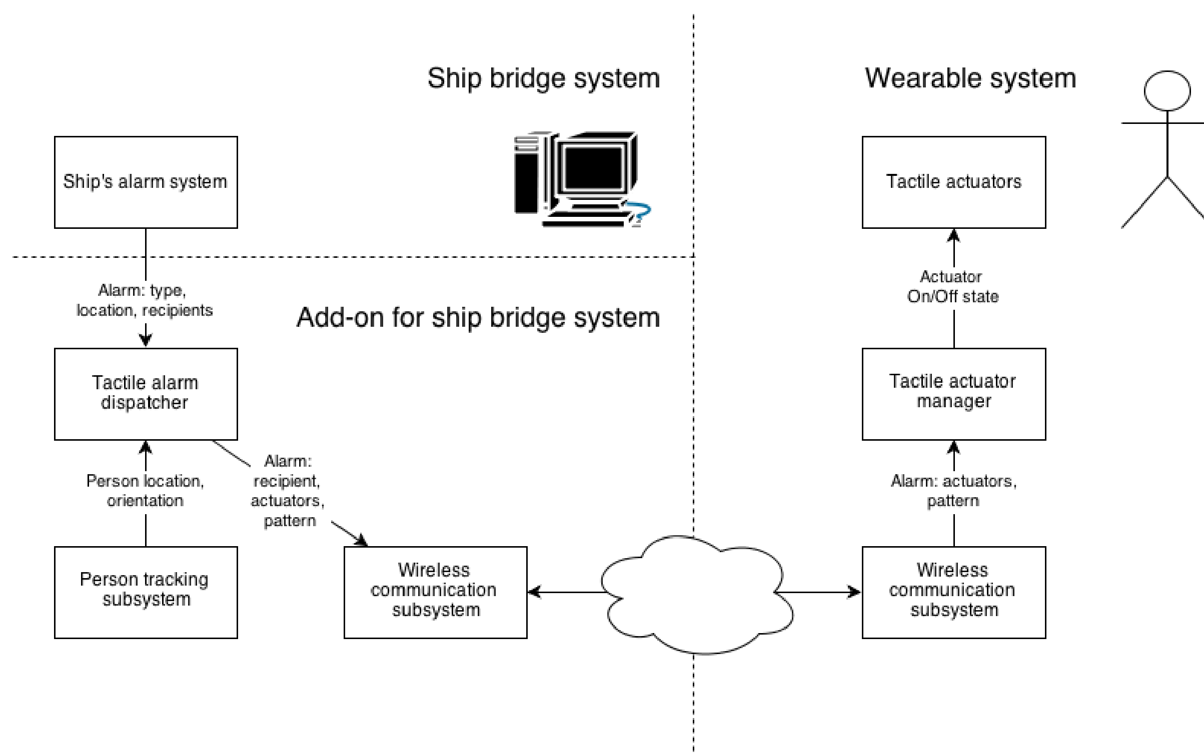

Figure 5.1: Tactile ship bridge alarm system architecture - wearable sensor and actuator device communicating with ship bridge automation system

follow the person 24 hours a day (perhaps also in bed, if it is comfortable enough), and ability to give accurate directions.

5.3.3 Sensor network aspects

The tactile alarm system presented here is a sensor-actuator network, although the first implementations might seem otherwise. Although the main focus of the system is actuation, not sensing, in further, more advanced revisions, the system would contain sensor modality, such as position and pose estimation with inertial sensors, in combination with external vision-based user tracking. In the experimental phase the author has assembled only one belt, yet for deployment at least two belts are required for maritime operators (such as dynamic positioning and anchor handling operators), one belt for the captain and optional belts for other crew members. Continuous connectivity would require a wireless base station and router infrastructure that is able to provide two-way communication with the mobile, wearable devices in the harsh environment of ships containing thick steel walls. Wireless communication is proved to be possible on passenger and offshore vessels[1].

[1]http://www.mtnsat.com/mtn-solutions/internet-wi-fi

5.3.4 System prototype

The author designed tactile belt as the first external, wearable tactile device for a ship bridge alarm system. The following sub-sections describe hardware and software components of the solution.

5.3.4.1 Hardware components

As can be seen in Figure 5.1, the device consists of three components: tactile actuators, actuator manager, and wireless communication. All these components are independent and can have different implementation as long as the interaction protocol is followed. For example, wireless communication can be implemented using WiFi, BlueTooth, ZigBee or other standards; AVR, MSP430 or other microcontrollers can be used as actuator managers; and different vibrating motors are supported.

The author has created a hardware prototype, shown in Figure 5.2. Its structural diagram is shown in Figure 5.3.

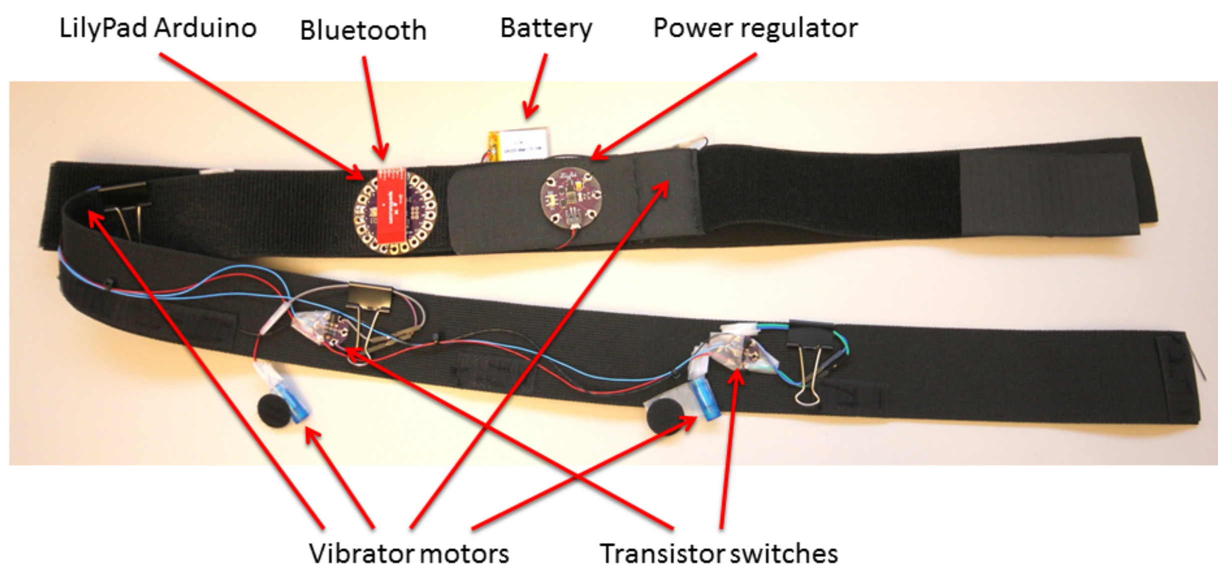

Figure 5.2: Tactile belt prototype - Vibrator motors, microcontroller, battery power source and wireless communication mounted on a stretchable material

The belt consists of the following components:

- Bluetooth radio module acting as a wireless bridge between the belt and external alarm system. Bluetooth Mate silver [1] used for the prototype, consisting of Roving Networks RN-42 Bluetoth Class 2 module [2].

[1] https://www.sparkfun.com/products/10393
[2] http://www.microchip.com/RN42

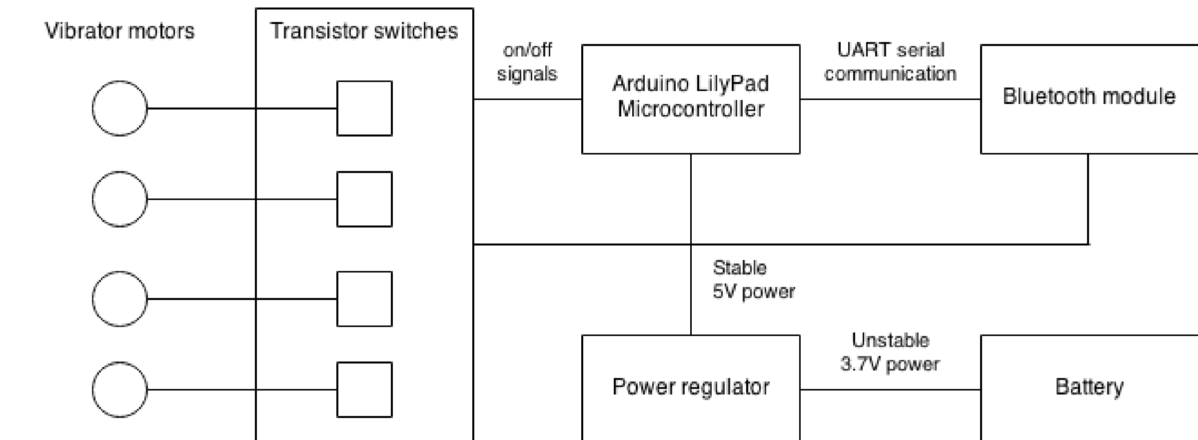

Figure 5.3: Tactile belt architecture - in addition to microcontroller, bluetooth module and power source, transistor-based switch circuit is used to drive high-current vibrator motors

- 4 vibrating motors generating tactile cues. They are located across the abdominal of the user: one motor in the front, one in back, one on the left side and one on the right side. Literature studies show that users can distinguish between 8 evenly spaced locations on a tactile belt [156], yet it is sufficient with four in the first scenarios. And the architecture is flexible additional motors can be added later if necessary. A switch circuit with a transistor is added for each motor so that it can be controlled by a microcontroller. Precision Microdrives 307-100 Pico Vibe 0mm-25mm vibrating motors [1] are used in the prototype with switch circuits consisting of BC368 NPN transistor [2], 1N4148 diode [3] and resistor mounted on a LilyPad Small Protoboard [4].

- An Arduino LilyPad microcontroller [5] acting as the manager: parsing wirelessly received messages and sending commands to motors.

- A Lithium Polymer (LiPo) battery powering the whole belt. A 400mAh battery weighting 9 grams (0.32 oz) [6] is sufficient to supply the system for about 8 hours. A 2000mAh battery (36 grams or 1.27 oz) [7] would last about 40 hours.

[1]https://catalog.precisionmicrodrives.com/order-parts/product/307-100-9mm-vibration-motor-25mm-type
[2]http://www.promelec.ru/pdf/BC368-NXP.pdf
[3]http://www.nxp.com/documents/data_sheet/1N4148_1N4448.pdf
[4]https://www.sparkfun.com/products/9102
[5]https://www.sparkfun.com/products/9266
[6]https://www.sparkfun.com/products/10718
[7]https://www.sparkfun.com/products/11238

- A power regulator module [1] transforming unstable 3.7V battery voltage to a stable 5V power source.

5.3.4.2 Software components

The author designed the software as master-slave (or client-server) system where tactile devices act as slaves/clients receiving commands from a central computer. In deployment the central computer is represented as a module in the ship bridge alarm system, while in test scenarios it can be any personal computer or any other device capable of connecting to the tactile device wireless network.

Client devices were programmed by the author using Arduino Integrated Development Environment (IDE), server application is developed in Java, using RXTX serial communication library[2].

The motors are activated by sending `MotorCommand` message from the server to client. The client responds with `Acknowledgement` message. If the server receives no `Acknowledgement` within a certain period of time after sending a `MotorCommand`, it should resend the `MotorCommand` message. Timeouts and number of retries are system specific and are not defined here.

5.3.5 Prototype conformance to design rules

In the following section the tactile system implementation is analyzed with respect to conformance with wireless sensor network software design rules proposed in this thesis. The analysis shows that the first prototype suffers from problems related to network lifetime, limited space coverage and deficient multitasking. These drawbacks can be mitigated by following design rules, as it will be shown in Section 5.3.6.

5.3.5.1 Communication

- **Sink-oriented**: satisfied. Although most of the communication actually initiates at the sink and transmission of data (commands) is in the direction from sink to wearable devices, Bluetooth is a master-slave technology. The computer is acting as the master with the role of the sink.

- **Powered motes**: satisfied. It is assumed that master device is always on and always received the message sent by slave devices.

[1] https://www.sparkfun.com/products/11260
[2] http://rxtx.qbang.org/wiki/

- **30 byte packet**: satisfied. The default packet size specified in Bluetooth RFCOMM protocol (implements serial communication link over Bluetooth wireless channel) is 127 bytes, configurable in the range from 23 up to 32767 bytes [157].
- **11 hop routing**: not satisfied. Single-hop communication is considered in the first prototype, no router infrastructure is used.
- **CSMA MAC**: satisfied. Bluetooth is able to operate both using asynchronous (CSMA/CA) and synchronous (TDMA) MAC protocols [158].
- **Custom protocol API**: not satisfied. The Bluetooth modules used do not allow changing MAC protocol.
- **Packet acknowledgement**: satisfied. Bluetooth RFCOMM provides a reliable communication channel [157].
- **IPv6 support**: not satisfied. Bluetooth uses local addressing scheme and RF-COMM profile does not support IP protocol.

5.3.5.2 Portability

- **TelosB support**: not satisfied. Arduino software (used in current implementation) supports only Atmels AVR-based microcontrollers.
- **Rapid driver development**: satisfied. Arduino system provides convenient interface for driver development in object-oriented C++ language and lots of examples are available on community forums.
- **Rapid platform definition**: not satisfied. Arduino is intended for use only on AVR-based microcontrollers.
- **802.15.4 support**: satisfied. Although Bluetooth uses a different protocol (IEEE 802.15.1), interface from Arduino board to Bluetooth module is UART. The Bluetooth module currently used can be replaced with an 802.15.4 XBee module and the Arduino program may be left unchanged.
- **AVR and MSP430 support**: not satisfied. Arduino OS and IDE are designed for Atmel AVR microcontrollers only.

5.3.5.3 Task scheduling

- **Low duty-cycle**: not satisfied. Arduino application is running in a 100% duty-cycle.
- **5 kernel + 6 user tasks**: not satisfied. The current Arduino application has a single thread of execution handling wireless communication and motor management.
- **Cooperative scheduling**: not satisfied. Arduino does not have any multi-task scheduling mechanism.
- **Preemptive scheduling**: not satisfied. Arduino does not have any multi-task scheduling mechanism.
- **Event-based scheduling**: not satisfied. Arduino does not have any multi-task scheduling mechanism.

5.3.5.4 Services

- **External storage**: satisfied. Arduino community provides source code for different external memory devices, including SD card.
- **File system**: satisfied. There are implementations of FAT16 and FAT32 file systems available for Arduino environment. Although FAT is a rather complex file system in the WSN context, it has a simple user interface on the Arduino side and is supported by any PC operating system, including Windows, Linux and MacOSX.
- **Time synchronization**: satisfied. Arduino time synchronization library is available supporting time pulses from different sources, including GPS.

Although none of these services are used in the deployment, libraries and examples are available if such need arises in the future.

5.3.5.5 User support

- **Base station example**: satisfied. Different Arduino application examples as well as source code for the PC side can be found in the community pages.
- **Popular sensor API**: satisfied. This particular application might need accelerometer and light sensors to determine persons position and environment. There are numerous drivers and source code examples available.
- **ADC API**: satisfied. There is an API for analog sensor reading in the Arduino OS.

- **Remote access**: satisfied. Bluetooth RFCOMM emulates virtual serial connection for data exchange. Remote reprogramming is not possible in this case. But it is also not very important - only a couple of network nodes are used and they are accessible during the development time.

5.3.6 Improvements by matching design rules

In this section the author identifies problems in the system prototype and discusses future improvements based on WSN software development design rules proposed in this thesis.

The following problems have been identified for the prototype implementation:

- **Short network lifetime**. The devices are not able to operate autonomously for the desired period of 7 days. There are multiple reasons for that, including energy-inefficient hardware and task scheduling.
- **No multi-hop communication support**. While single-hop communication is reasonable for tactile alarm dissemination in a single room (ship bridge) it prohibits the implementation of alarm forwarding to watch officers in other facilities.
- **No multitasking**. One can implement all required processes (motor control, data reception, data transmission and sensor sampling) in a single thread, yet it involves creation of a state machine with inefficient and error-prone polling strategies.

The author analyzes improvements that are suggested by the proposed design rules in the remainder of this section.

5.3.6.1 Network lifetime extension

There are two modes of expected system operation:

- During intense operations, where tactile device might inform the operators of critical information, low latency is important (in millisecond range). Therefore 100% radio duty-cycle is expected here.
- During other time (actually, most of time) crew members are in idle mode, when nothing significant is happening. They must be warned in case of alarm, yet the acceptable latency is much higher (might be several seconds).

Most energy is spent in radio listening mode. Customized MAC protocols (*design rule#6*) that allow changing radio duty cycle can help to reduce energy consumption significantly. For example, if the radio transmission is activated every 5 seconds for a

250ms period (it takes around 100ms to send a 46-byte packet [159], 250ms is enough for two-way communication), it results in a 20% duty cycle.

The current Bluetooth module does not allow control of MAC protocols. Therefore more efficient radio module must be selected. In addition, the Arduino board with AVR ATMega328 microcontroller is also not the best option in terms of energy efficiency it consumes around 25mA in active mode, and additional 25mA for Bluetooth radio, the total consumption of the platform is more than 50mA or less than 8 hours of operation from a 400mAh battery.

Vibrator motor energy consumption cannot be accurately predicted without a particular scenario. The motors will be active in a very tiny fraction of time, the duty cycle will be very low. Let us examine an example scenario where 4 motors are used, each of them consumes 50mA of energy in active mode. The operators are active 8 hours daily performing operations where alarms may be raised once every 5 minutes and the motors are active for 1 second on every alarm. That means a 96 seconds of active motors during the 8 hour operation or 5.33mAh of total energy consumed. During the inactive period of the day the probability of an alarm is low, let us approximate it to one alarm every day. However, the motors will be active longer on each alarm, let us define the activity period 60 seconds in this case the person must react and turn the alarm off in one minutes time. That leads to 60 seconds of motors in active mode or 3.33mAh of energy consumed. Taken together, less than 9mAh of energy is consumed daily for the motor operation or less than 0.375mAh of average consumption. Although this example uses multiple assumed constants, it shows that the motor energy consumption in a realistic scenario is insignificant, compared to consumption of the rest of the system.

Selection of an energy-efficient wearable sensor-actuator node increases the lifetime dramatically. Let us take a TelosB-compatible platform with MSP430F1611[1] microcontroller and CC2420 radio, such as TMote Sky, as an example. The whole platform consumes 20-23mA during active radio transmission or reception. With a 20% duty-cycle that would result in less than 5mA average consumption. It is tenfold increase in energy efficiency compared to existing implementation. To conclude, a solution that supports custom MAC protocols (*design rule#6*), TelosB-compatible platform (*design rule#9*) and low duty-cycle (*design rule#14*), would lead to significant lifetime extension.

In addition, it is important to be able to experiment with multiple different platforms and select the best alternatives based on empirical evidence. Rapid porting and driver development (*design rules #10 and 11*) are important in that matter. To comply with these rules, Arduino software should be replaced with a solution more appropriate for

[1]http://www.ti.com/product/msp430f1611

porting to new platforms and providing wider set of supported platforms. Contiki OS would be a good candidate: the solution can be incrementally ported to Contiki OS keeping the same initial hardware and than changing hardware component-by-component as necessary.

5.3.6.2 Multi-hop communication

To implement a deployable system, alarm dissemination is required also outside the ship bridge room, and a 24-hour stable operation is required. Multi-hop communication (*design rule#4*) between the alarm generation system and tactile wearable devices is essential part of this requirement. The solution can be implemented in multiple different ways: either the conventional ship automation systems network (TCP/IP or other) is used to create a backbone network and connect tactile devices using gateway nodes attached to each backbone network router, or a mesh network of wearable devices and corresponding sensor network routers (802.15.4) can be installed on the ship, connected to the automation systems network using a single (or multiple redundant) gateway nodes.

5.3.6.3 Multitasking support

There are multiple logical tasks running concurrently on the wearable device: motor control, data reception, data transmission and sensor sampling (no sensors attached at the moment, but could be required in future deployments). Support of multi-tasking by providing API for separate thread creation (*design rule#15*) is necessary due to different aspects. First, it is correct to separate and encapsulate threads with different responsibilities and resources. It is logically more correct and makes the code easier to maintain and expand. Second, correct multi-tasking can improve the efficiency of the application in terms of time-sharing threads wait when they have no operation to perform and start running whenever the expected event has occurred. Fully accurate multitasking is not achievable on a single-processor microcontroller, yet the idle-time can be minimized.

Selection of scheduling techniques depends on task characteristics. If some of them are time-critical (MAC protocol) while others may be time-intensive (data processing), preemptive scheduling is required (*design rule#17*). If there is no intensive data processing, only command execution and sensor data reports, cooperative scheduler (*design rule#16*) is sufficient and will have less overhead on average. As the whole system is event-driven, event-based scheduling with configuration on callback function (*design rule#18*) would be very efficient in terms of system performance, yet it might be more difficult for the programmers if the system grows more complicated during its evolution.

5.3.7 Use case summary

This section describes a wireless sensor network use case where the author created a prototype implementation of wearable wireless device. The author identified several problems, including short network lifetime, limited communication abilities and problematic source code design and maintenance. Then he showed that following several of the WSN design rules proposed in this thesis can make significant improvements and can solve the identified problems. To conclude, this use case showed that the design rules are applicable for WSN quality assurance as a diagnostics checklist and also solution guide.

5.4 Summary

In this chapter the author showed that proposed design rules are a valuable tool for improvement of WSN software at different development phases. The rules can be used to identify drawbacks as well as guidelines for design and further development of existing systems. In addition, the rules are applicable to WSN software at different levels: from operating system to particular applications. Rule evaluation for new WSN software design will be analyzed in the following chapter.

6 New operating system design according to rules

The previous chapter discusses how proposed design rules can be applied to evaluate existing WSN software solutions and conclude on future improvement directions. This chapter describes how the same design rules can be used during the design and implementation processes of a new operating system for wireless sensor networks.

The Object-Oriented-MansOS, or OOMOS for short, is an operating system that is incrementally built based according to the design rules proposed in this thesis. At the moment of writing this chapter, OOMOS is in a work-in-progress state. Nevertheless, OOMOS is applicable to simple WSN applications already at the current stage, and this chapter shows the next steps for OOMOS to become a widely-applicable WSN operating system.

The author of this thesis has designed and implemented OOMOS individually, therefore it is considered solely the author's contribution. In addition to the main goal of evaluating design rule applicability for new WSN OS design, an additional challenge was set for OOMOS. As the title states, OOMOS is a proof-of-concept showing that object-oriented programming approach can be used in WSN operating systems and applications. To the best of the authors knowledge, there is no other fully-functional object-oriented WSN operating system.

The rest of this chapter will describe OOMOS' advantages over MansOS (Section 6.1), give introductions to object-oriented programming and its adaptation to WSN domain (Section 6.2), describe implementation of OOMOS according to design rules (Section 6.3), evaluate OOMOS in terms of portability and performance (Section 6.4) and discuss future evolution of OOMOS according to the rules (Section 6.5).

6.1 OOMOS' advantages over MansOS

To the best of the author's knowledge, OOMOS is the first object oriented operating system for wireless sensor networks. As such, it is a proof-of-concept example that object-

oriented design is possible and feasible for WSN OS.

Object-orientation comes with overhead in terms of code size and execution speed. OOMOS advantages compared to MansOS are related to usability aspects. OOMOS provides a new way of designing and structuring the code with a goal of increased familiarity, clarity, and modularity:

- *Familiarity.* Object oriented programming paradigm is very popular amongst programmers - 4 of 5 top programming languages (according to TIOBEs research [160]) provide object oriented programming as the main paradigm. Familiar environment is very important to attract new sensor network programmers, as it decreases the familiarization time [161].

- *Clarity.* OOMOS provides a way to explicitly define provided and used interfaces for components similarly to TinyOS, yet in a more familiar and straight forward way. Instead of writing additional configuration files with custom syntax, component wiring in OOMOS is implemented using pure C++ language `set()` and `init()` routines.

- *Modularity.* OOMOS implements MCU support code as set of modular components, such as ADC module, SPI module, UART module. These components can be reused to adapt another MCU that supports the particular module. Such modular approach increases porting speed of multiple MCUs from the same MCU family as they usually share many common modules.

The goal of implementing OOMOS is to create an object-oriented WSN OS according to proposed design rules while maintaining portability and performance at a reasonable level. It is expected that object-orientation will add overhead to the system. Nevertheless, evaluation will show(Section 6.4) that performance of compiled C++ code is at usable level while providing the benefit of familiarity to object-oriented programmers.

6.2 Object-oriented programming for WSNs

Computer programming can be done in many different ways. The term *programming paradigm* is used to describe a specific style of designing and programming computer systems. Object-oriented programming (OOP) is one of the popular paradigms, that encapsulates data and methods in entities called *objects*.

Although all widely used operating systems (including Linux, Windows and MacOS) are procedural, there are several advantages of object-oriented operating systems. See Section D.1 in Appendix for more details. Object-oriented operating systems have been

proposed previously, including ChoicesOS [162, 163] as an academic example and Haiku OS as a commercially available OS [164, 165].

In addition to conventional OOOS general advantages (portability, maintainability and extensibility) wireless sensor network OS design, using object-oriented principles, provides some benefits that should be noted explicitly:

- More flexible code reuse in MCU-family hierarchies. For example, MSP430F1611 MCU can inherit code common to the whole MSP430 architecture, MSP430 Series 1 family chips, and it can use modules common to multiple chips, such as TimerA3 and USART1 modules.
- Convenient interface for definition of loosely coupled components, i.e. usage of conventional classes and objects is a simpler alternative of interdependent object description interface, provided by TinyOS operating system.
- Programming to an interface, not fixed implementation, allows to switch actual implementations either at compile- or run-time. This principle can be used to choose different data processing algorithms, networking protocols or communication hardware interfaces (data sinks).
- Data encapsulation allows object to store internally all constants and variables it depends on. Although, this approach uses significantly more memory, compared to macro-constant usage in specified header files, it allows more dynamic module usage. For example, multiple instances of the same chip (radio, flash memory) may be used on the same platform.

6.3 OOMOS implementation

The author chose C++ as OOMOS implementation language, due to its effectiveness and popularity among programmers around the world. An alternative would be to use Java. However, Java uses virtual machine paradigm, poses high overhead on system performance, and Java for WSNs is not a novelty [58, 60, 166].

The following subsections describe how OOMOS was designed according to design rules, proposed in Chapter 4.

6.3.1 Portability

The author designed OOMOS as a wireless sensor network operating system right from the beginning. Therefore it targets low-power platforms suited for WSN applications. The

author chose TelosB as the first hardware platform according to *design rule#9*. Support for CC2420 radio with 802.15.4 standard communication was implemented according to *design rule#12*. To conform with *design rule#13*, Arduino support was included as a representative of AVR-based platform. Zolertia Z1 was chosen as a platform to test portability and code reusability of the OS (*design rule#11*).

The author implemented device drivers in platform-independent manner, to increase code reusability. Although such requirement may seem trivial and self-evident, it is not always met in other operating systems. For example, in Contiki OS part of device drivers are platform-specific, including ADXL345[1] acceleration sensor driver for Zolertia Z1 platform, that uses platform-specific pin operations.

Device drivers are object-oriented with classes representing devices. Existing procedural code is reused, wrapped in C++ classes. Although such approach requires more effort at the beginning, more reusable code pieces are created, representing individual modules of devices.

Object-oriented approach is useful for drivers of generic modules, that are specified by subclasses. Each subclass may execute the same code, defined in parent class, by using only descendant-class-specific parameters. Strategy pattern is used here. Example class: `MSP430_USART`, that provides routines for its child class modules: `MSP430_USART0` and `MSP430_USART1`.

Similarly to approach used in TinyOS [48], OOMOS is composed of interfaces and components (or objects) implementing these interfaces. In contrast to TinyOS, where nesC language-specific construct are used to wire components and interfaces, OOMOS uses standard C++ languages primitives for that matter. Each component has a set of provided and required interfaces. Provided interfaces are implementations of functions that can be used by other objects. Required interfaces are functions that may be called by this object and must be implemented by other objects. Interface mechanism is an important part of OOMOS increasing interoperability between objects.

Interface primitives provided in programming languages, such as Java, are not available in C++. Therefore interfaces are implemented in OOMOS as abstract base classes with pure virtual methods - declared methods without implementation. An example `ILogStream` interface is shown in Listing D.1 (see Appendix).

The same interface may be implemented by multiple objects. For example, `ILogStream` interface is implemented by UART (Listing D.2), radio (Listing D.3) and external flash memory modules. A comfortable feature of OOMOS - interface hierarchies can be created - an interface may implement another interface. For example, `IUART` interface for

[1] http://www.analog.com/static/imported-files/data_sheets/ADXL345.pdf

UART module handling automatically implements `ILogStream` interface (Listing D.2). That means, that each UART module, providing `IUART` interface, automatically provides `ILogStream` interface for logging data streams. An object may provide multiple interfaces by using multiple inheritance. For example, CC2420 radio chip (Listing D.4) provides `IRadio` interface - it can be used as a radio chip on multiple platforms. Simultaneously, it provides `IEventHandler` interface - CC2420 listens to General Purpose Input/Output (GPIO) pin interrupts.

A new provided interface may be added for a class by simply specifying additional parent class to inherit from, and implementing its virtual functions. Abstract interface class names are prefixed by capital letter `I` to separate them from object classes.

Required interfaces for each class are specified by using class variables - pointers to interface objects. For example, CC2420 radio chip requires `ISPI`, `IMCU` and `IGPIO` interfaces (Listing D.4). These pointers are initialized in `init()` functions, that are class-specific. While it is not possible to detect in compilation time, which interface dependencies are not met, `init()` functions return error code `ERR_MISSING_COMPONENT` of type `err_t` in case of missing dependency. An example of initialization of interfaces for TelosB platform is shown in Listing D.5.

Microcontroller is the core of each WSN hardware platform, and its support constitutes the most part of platform code. To provide flexible portability and code reuse (*design rule#11*), MCU support code is divided into three layers:

- *MCU architecture* - code common to the whole architecture of MCUs, such as, AVR or MSP430. This layer contains routines common for all MCUs of this architecture, and MCU components (ADC, timers, etc) that are reusable in multiple MCUs of the particular architecture, yet are not specific to a particular MCU model or family.
- *MCU family* - a *family* or *series* of MCU chips, that have common modules and shareable code for all members of the family. Example families include MSP430 Series 1, MSP430 Series 2.
- *Particular MCU model* - a chip-specific code for a particular MCU. Examples: MSP430F1611, ATMega328p.

`IMCU` interface is an abstract class providing functions that should be available on most MCUs, such as timer counter functions, GPIO pin, ADC and watchdog module handling. MCU architecture code provides functions either common to the whole architecture, or modules that are available on multiple MCUs of a particular architecture across multiple families. Examples of such modules include `MSP430_TimerA3` module available

on MSP430x1xx, MSP430x2xx and other family MCUs; and AVR_ADC module present on multiple AVR-architecture MCUs.

Such code distribution across multiple layers gives flexible porting and code reuse capabilities. For example, to add support for a new MSP430x2xx family MCU, MSP430F2274, only a few classes must be implemented: MSP430_ADC10 module and the MCU class itself, see Figure 6.1.

The code separation in three layers is similar to approach used in MansOS. However, OOMOS adds improvements in code portability by introducing *MCU family* concept and providing MCU family classes with code reusable for all members of a particular family.

OOMOS takes a slightly different approach in system initialization, compared to MansOS. MansOS kernel sequentially initialized all the modules that are used in a particular application: Light Emmiting Diodes (LEDs), ADC, I^2C, radio, etc. In contrast, OOMOS calls only one function: `platform.init()`. This `init()` function must be provided by each platform and contains initialization sequence, specific to each platform. An alternative approach in OOMOS would be implementation of default init sequence in the `IPlatform` base class, and delegation of single module initialization steps to ancestor classes, similarly as in MansOS. However, OOMOS approach provides more flexible approach for different platforms. If a common initialization pattern is discovered for a set of platforms, an abstract base class between `IPlatform` interface and specific platform classes may be introduced to provide common framework for all the platforms with similar initialization sequence.

6.3.2 Scheduling

The author designed OOMOS for low-power applications (*design rule#14*) and provides effective scheduling techniques: cooperative protothread scheduling [137] was adapted from Contiki OS [47] (*design rule#16*), similarly to MansOS. See Section C.1.2.1 in Appendix for more details on the scheduler. No object-oriented wrapper classes were added, protothread API is used in its standard form. Reimplementing it in object orientated fashion would not simplify the syntax, yet it may require additional testing and introduce new errors. In addition, usage of the standard protothread API in OOMOS and MansOS allows easier application porting and sharing between all three operating systems: Contiki, MansOS and OOMOS.

In addition, the author identified the following technical substantiation for cooperative scheduler:

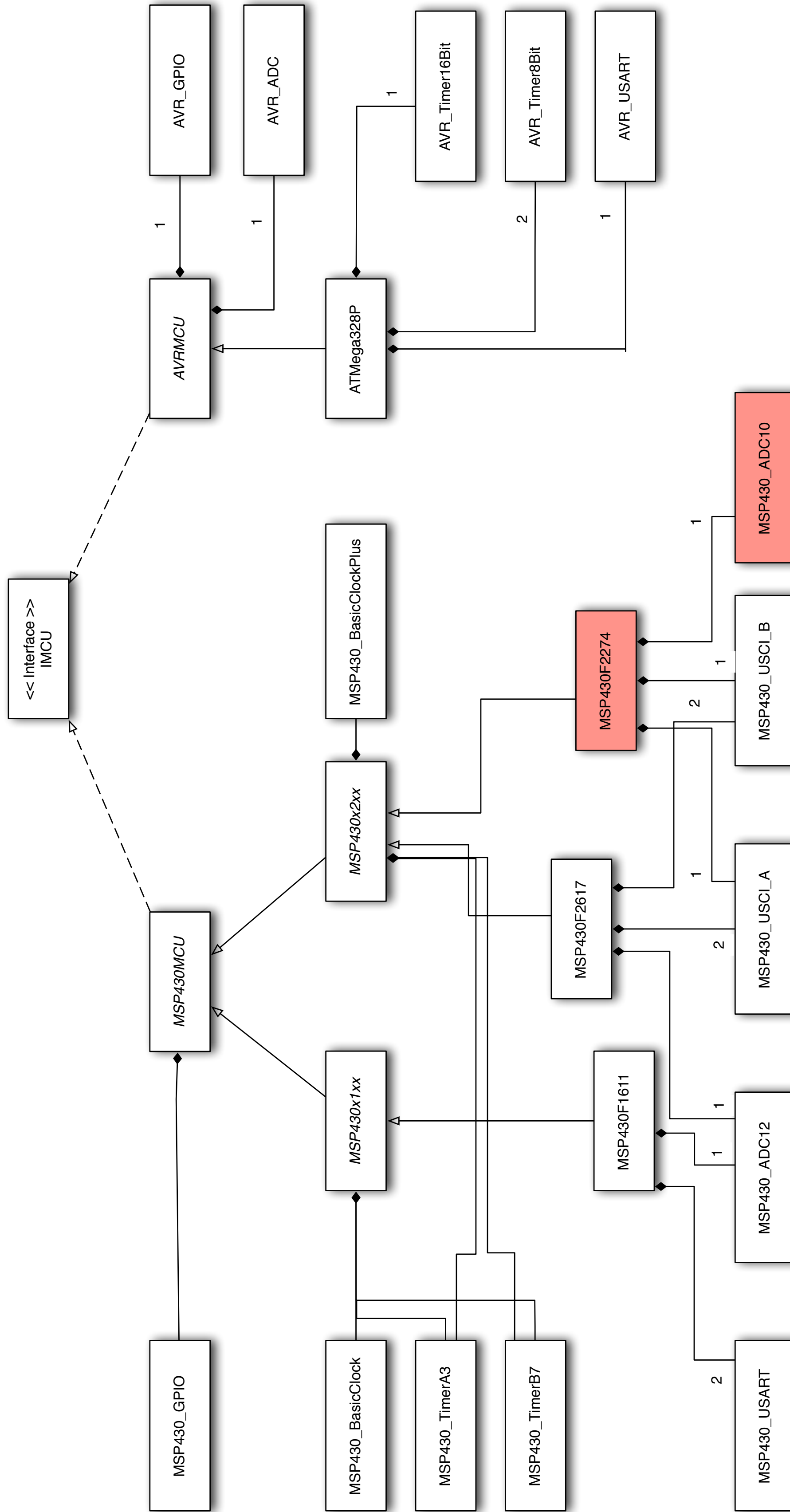

Figure 6.1: OOMOS MCU class diagram with new MCU supported - Addition of a new MSP430x2xx family MCU MSP430F2274 in OOMOS requires addition of only two classes: ADC10 module and the MCU itself

- As the evaluation of existing solutions shows (Contiki radio packet reception, Section 6.4.2.3), cooperative scheduling is able to provide high performance in handling time-critical tasks.
- Cooperative scheduling with single memory stack simplifies memory sharing between processes and decreases stack overflow probability.
- Cooperative scheduler requires less platform specific code: only some MCU timer routines. No context switch code is required.
- Preemptive scheduler can be added later as an option.

No particular limits on concurrent proto-thread count were set in OOMOS. Memory resources are sufficient to support at least 5 kernel and 6 user tasks, as required by *design rule#15*. In addition to cooperative scheduling, users can directly use callback functions of device drivers, including UART serial port and radio chip. Although the author does not recommended it, event-driven programming paradigm is supported (*design rule#18*) in OOMOS.

6.3.3 Services and API

Although hardware platforms for sensor networks are very different, operating systems should provide some common components and routines for all platforms to simplify user application development. Whenever a particular platform does not support a very common feature, compiler should the user of this fact.

The author has divided platform API (Listing D.6 in Appendix) into three parts:

- Mandatory API - functions and components required on each WSN platform. Examples include platform initialization routine `init()`, called by the kernel, delay and timer routines.
- Very common, yet optional API - functions that should be provided by majority of platforms, yet may be inaccessible on some nodes. Examples: ADC (*design rule#24*), radio modules, LEDs, light, temperature and accelerometer sensor functions (*design rule#23*) and external memory handling (*design rule#19*).
- Optional, platform-specific routines, such as routines for reading specific sensors (seismic, gas, barometric pressure).

The author used an abstract interface `IPlatform` as a base class for all WSN hardware platforms to implement the three API layers. It declares pure virtual functions for all mandatory functions. These must be implemented by all platforms. OOMOS specifies virtual functions with default, empty implementations in `IPlatform` interface to provide common feature interface to user applications. These default implementations are overridden by all platforms supporting the particular feature. For example, `readLight()` function returns 0 by default (see Listing D.6). It is implemented by calling appropriate sensor device readout function on all platforms supporting light sensor. OOMOS implements fully optional functions as simple, non-virtual functions in each platform's class.

To support partially optional, common functions, two alternative approaches could be used:

- Specification of all required functions as pure virtual. It is not desirable, as it would require all platforms to implement all functions, even if they are not supported. It would lead to larger code. And introduction of new functions in `IPlatform` interface would require modification of all platform classes, which is not a desirable situation.
- Removal of partially optional functions from `IPlatform` interface. While it could save code space, such approach would make user applications more platform-specific and not allow to compile the same application on all platforms, even if part of used components are not available on the platform. For example, application may check availability of sensors on the platform, sample and report only available sensor values.

6.3.4 Summary

OOMOS is an experimental OS with the goal to build an object-oriented WSN OS according to design rules proposed in this thesis. The development process of OOMOS is incremental and there are design rules that are not satisfied in the current stage. The main missing part is full networking protocol stack, designed according to specific WSN needs and proposed design rules. The gaps will be identified and suggestions on future directions will be given in Section 6.5.

6.4 OOMOS evaluation

The author has developed OOMOS as an example how an object-oriented WSN operating system can be built according to proposed design rules. MansOS and OOMOS are

operating systems with high portability in mind. This section evaluates how conformance to design rules has helped MansOS and OOMOS in reaching higher code reusability.

In addition, it is important that WSN operating systems possess reasonable performance. Evaluation shows, that Contiki and TinyOS are superior in some performance aspects, yet MansOS and OOMOS still have performance at a level that is efficient for practical applications.

Being a modification of MansOS, OOMOS must be evaluated together with MansOS. Whenever applicable, MansOS and OOMOS are compared to two typical WSN operating systems: TinyOS and Contiki. TinyOS represents the most popular choice among WSN users, and is built with extreme resource efficiency in mind. Contiki, on the other hand, focuses more on user friendly programming interface. MansOS and OOMOS strive to achieve high user friendliness and portability while maintaining a reasonable degree of performance and resource efficiency. As the results reveal, in some cases MansOS and OOMOS are more resource-efficient compared to TinyOS, and in most cases have higher portability compared to both Contiki and TinyOS.

6.4.1 RAM and flash memory usage

To be usable for actual WSN applications, operating systems must have size requirements that can be met by existing hardware platforms. This section analyzes program code and RAM memory requirements of MansOS and OOMOS operating systems.

To assess memory usage, the author used test applications:

- `ADCPerformance` application, that samples ADC channels and outputs sampling rate.
- `RadioTxPerformance` application, that sends radio packets as fast as possible, and outputs transmission rate.
- `RadioRxPerformance` application, that receives packets transmitted by another sensor node, and outputs packet reception rate.

All three applications use UART serial data line for debug output, while ADC module is used only in the `ADCPerformance` application and radio module is used in both remaining applications: `RadioTxPerformance` and `RadioRxPerformance`.

In addition the author analyzed a simple LED `Blink` application, that used only LED module. It is usable to assess minimal size requirements of operating systems.

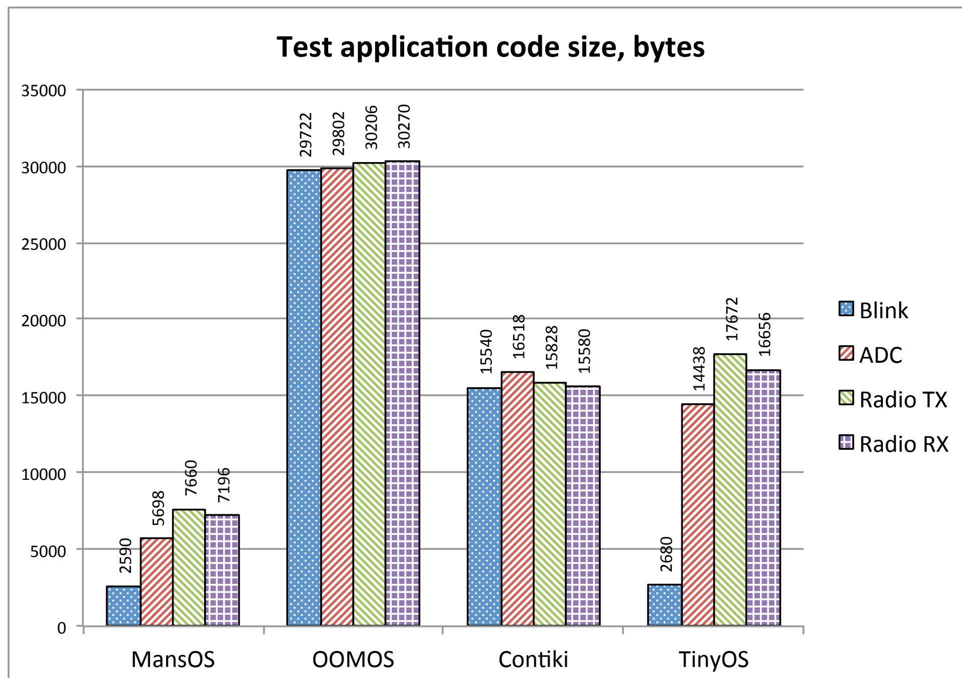

Figure 6.2: Test application program code size comparison in MansOS, OOMOS, Contiki and TinyOS - MansOS and TinyOs provide highly dynamic code size due to modularity, OOMOS object-oriented programs larger in size

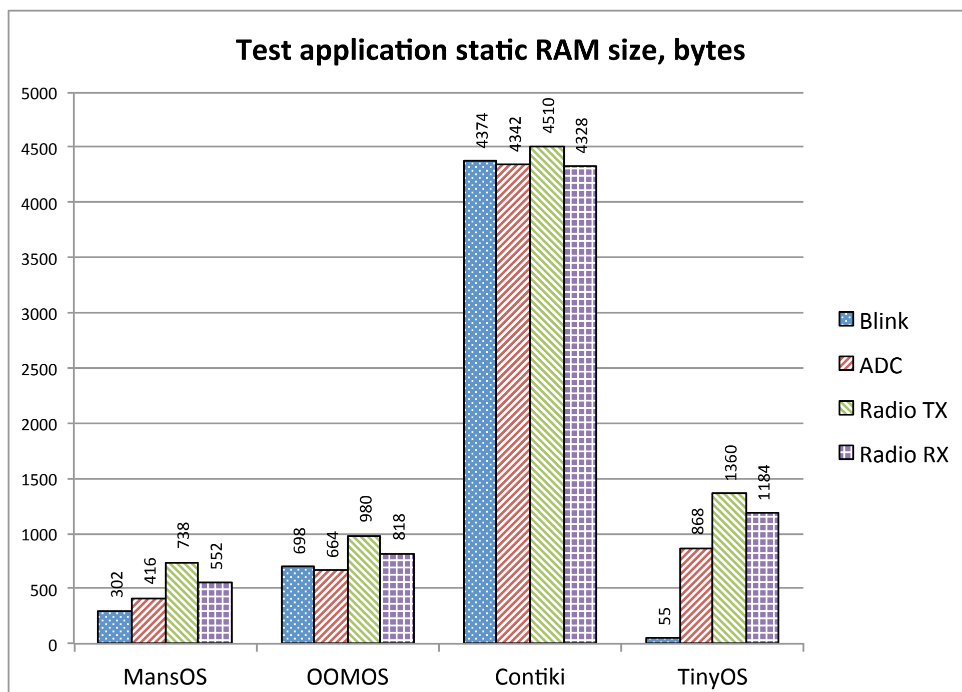

Figure 6.3: Test application static RAM size comparison in MansOS, OOMOS, Contiki and TinyOS - OOMOS RAM usage higher than in MansOS, yet significantly lower than in Contiki's default configuration

The author developed the same applications in four operating systems: MansOS, OOMOS, Contiki and TinyOS. Program code size and static RAM memory usage was analyzed. Results are shown in Figure 6.2 and Figure 6.3.

Multiple conclusions can be drawn from here:

- Program code size adapts very dynamically in MansOS (and also in TinyOS) according to features used in particular application. Unused modules are switched off and not included in the binary application image.
- MansOS code size is remarkably smaller compared to TinyOS and Contiki.
- Object-oriented code in its unoptimized form produces large program code binaries. While OOMOS application code size is small enough to fit in the flash memory of MSP430F1611 microcontroller (used on TMote Sky), optimizations of code size should be investigated, to make OOMOS usable for platforms with smaller program flash memory (see Section 6.4.3).
- MansOS has very low RAM memory footprint.
- OOMOS RAM footprint is higher compared to MansOS, yet it is still lower compared to Contiki and TinyOS. Therefore optimizations in OOMOS should be concerned more on saving program flash memory, than RAM size.
- Contiki uses large static RAM memory blocks, even in the default configuration. The reason - Contiki has multiple services, that use large memory buffers and that are also running, even when not used. Optimizations in this direction are possible - ability to turn off unused services in Contiki.

6.4.2 Performance

The author identifies three WSN application tasks as time-critical, where execution time is important:

- Sensor sampling. Some sensors, such as microphone or accelerometer, may require high-frequency sampling. Therefore sampling rate may be important.
- Data transmission over radio. Although sensor networks rarely use high-bandwidth communication, data transmission in short bursts is typical and ability to transmit several packets in a short time may be important. In addition, radio communication consumes most of used energy, therefore minimizing communication may extend network lifetime.

- Data reception over radio. If packets are transmitted at high rate (even if overall packet count is not large, yet more than one), it is important to be able to receive and buffer the burst of packets on the receiving node. Otherwise either information is partially lost, or request to resend lost data must be issued, diminishing the benefit of high-speed data transmission.

The author executed all three tests on TMote Sky platform [32], having MSP430F1611 MCU with advanced built-in ADC module, providing high frequency sampling, and TI CC2420 radio chip [8], widely used on WSN platforms. Cooperative scheduling with protothreads was used on MansOS and OOMOS to be comparable with cooperative schedulers used in Contiki and TinyOS.

6.4.2.1 Sensor sampling performance

The author chose analog sensor sampling using ADC as a test application for sampling performance analysis. The wide range of digital sensors vary a lot, therefore it is difficult to choose a typical representative of digital sensors. In addition, some sensors may have a sampling latency due to characteristics of the sensor itself. Therefore ADC sampling was chosen as a representative example of sensor sampling interface, with relatively stable characteristics for wide sensor range.

To measure sampling rate, the author developed a test application with consistent application logic in all four operating systems: MansOS, OOMOS, Contiki, TinyOS. It contains two test modes: (1) continuous sampling of a single ADC channel; and (2) sampling of three consecutive ADC channels consequently. The first mode was expected to have higher sampling rate, as the ADC module is not required to switch channels between each two samples. One test run is 30000 consecutive sensor samples. System uptime is checked after each test run, and sampling rate is calculated accordingly. The author analyzed 30 test runs for each test mode, and chose median of all sampling rates. Results are shown in Figure 6.4.

MansOS performance is the highest in single-channel mode, while in multi-channel mode it is 30% lower than Contiki's performance. The reason is Contiki sensor interface implementation for TelosB platform, that uses MSP430 ADC module specifics to sample all required ADC channels consequently, even when a single channel is required. TinyOS performance is high in single-channel mode, yet it is very low (under 1KHz) for multi-channel sampling. While `ReadStream` interface with batch operation support at high sampling rate was used for single-channel scenario, the slower `Read` interface had to be used for multi-channel scenario.

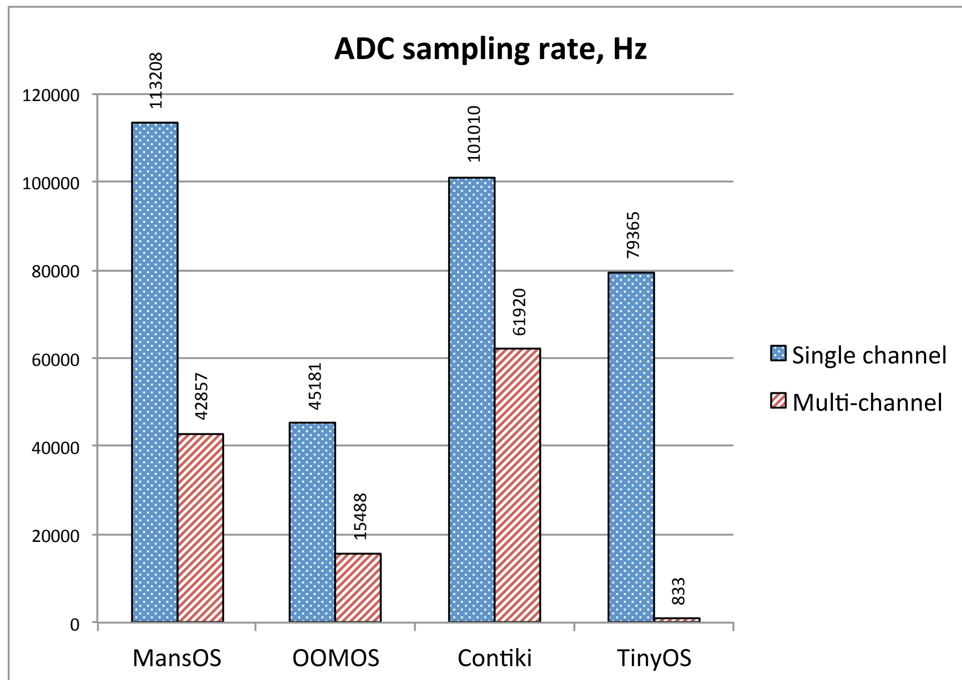

Figure 6.4: ADC sampling performance of MansOS, OOMOS, Contiki and TinyOS - MansOS performance is comparable to Contiki, and higher than TinyOS

Results show, that introduction of object-oriented approach in OOMOS has reduced performance of ADC sampling. Nevertheless, it is still on a level usable for majority of WSN applications, including single-channel audio sensor sampling at 45KHz or 3D accelerometer sampling with frequencies up to 5KHz (per channel). Built-in ADC sampling is not usable for demanding scenarios, such as seismic and acoustic data in volcano monitoring [3, 106]. External ADC modules with higher frequency are frequently used in such scenarios.

6.4.2.2 Wireless data transmission performance

The author used a test application with equivalent logic on all four operating systems (MansOS, OOMOS, Contiki and TinyOS) to evaluate radio transmission performance. Communication in the PHY layer was used to exclude networking protocol imposed delay. However, in TinyOS components of the communication stack are tied so closely, that it is impossible to access the PHY layer without rewriting most of TinyOS components, that would lead to a new operating system, that is not TinyOS anymore. Therefore Active Message (AM) layer was used. Therefore it must be kept in mind, that results of TinyOS could show lower throughput due to additional overhead of higher layer protocols. An alternative would be to use AM layer communication in all four operating systems.

However, it was not available in other OSes, therefore selection of already available and well-tested communication primitives was used. The AM layer in TinyOS adds 11 bytes of meta-data fields in addition to the packet, therefore it was not able to send 2 byte packets. For TinyOS, only 16, 40 and 120 byte packets were sent, were protocol meta-data fields were counted in the packet size (actual payload size: 5, 29 and 109 bytes).

The test applications executed four test modes in a continuous loop. The test node was sending packets with different sizes in each mode: 2, 16, 40 and 120 byte packets respectively. One test run was transmission of 1000 packets. System uptime was checked after each run and transmission rate calculated accordingly. 30 test runs were analyzed for each test mode, and median of all transmission rates was chosen. Resulting packet throughput (packets per second) is shown in Figure 6.5, and bandwidth (Kibibytes per second) is shown in Figure 6.6.

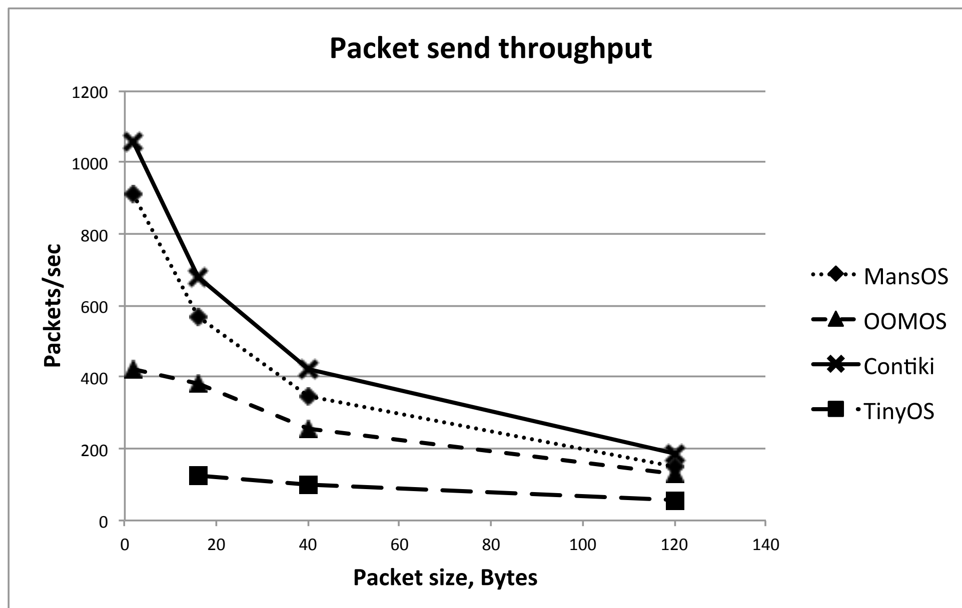

Figure 6.5: Radio transmission throughput dependance of packet size, packets/sec - Contiki has the highest transmission rate at Physical (PHY) layer, followed by MansOS. OOMOS performance is lower, yet the difference decreases with increasing packet size. TinyOS (AM Packet layer) performance is lowest.

Contiki has the highest radio transmission throughput, followed by MansOS. MansOS lower bandwidth might be explained by more generic SPI bus layer used, while it is MSP430 architecture-specific in Contiki. OOMOS transmission is lower than MansOS, due to object-oriented driver implementation, that requires more function cascades and virtual function calls in places, where plain C macros are used in Contiki and OOMOS. TinyOS performance is significantly lower, compared even to OOMOS. Partially it might

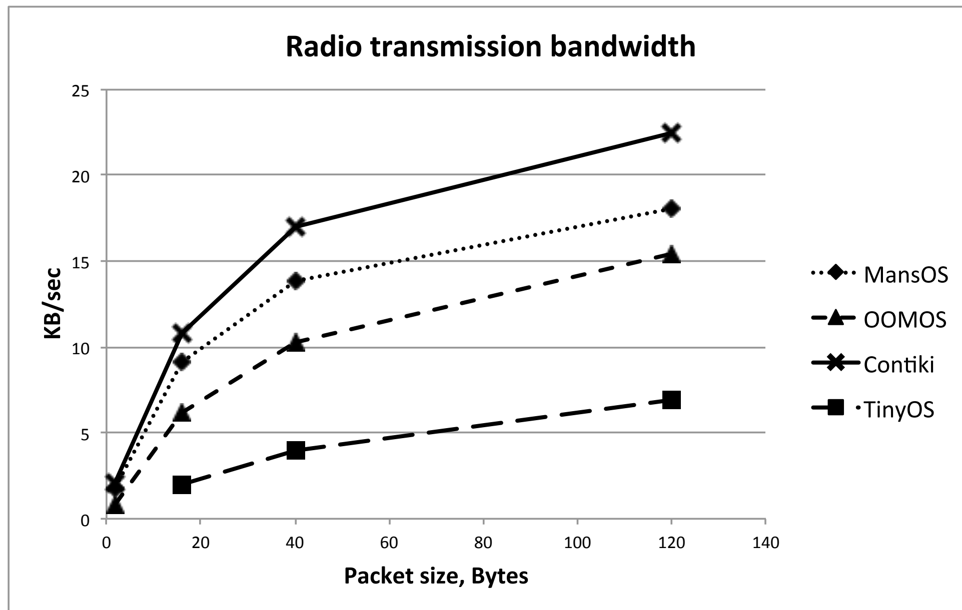

Figure 6.6: Radio transmission throughput dependance of packet size, KiBytes/sec -

be due to AM protocol stack overhead. Nevertheless, it shows that TinyOS does not always comply with its main objective to be a high-performance, low-overhead WSN OS.

The author's conclusion from test results - OOMOS object-orientation imposes additional overhead in data transmission, therefore, performance may be lower in applications requiring transmission bursts with high bandwidth, such as image sensor data transmission or data muling in delay tolerant networks [167]. In applications without such bandwidth requirements Contiki, MansOS and OOMOS will perform comparably well, as the packet transmission time difference is not more than a couple of milliseconds. It could be harder to implement high-precision time synchronization protocols in OOMOS.

6.4.2.3 Wireless data reception performance

High bandwidth data transmission becomes useless, if the receiving part is not able to process all the transmitted packets in time. The author performed additional radio reception (radio RX) tests to test OS ability to quickly buffer packets received over radio. An application with identical logic was used on all four operating systems. A 16-bit counter was analyzed in received packet, sent by a sensor node with radio transmission test application. The same operating system was used on both nodes. For example, when MansOS application was running on the transmission node, MansOS application was also running on the reception node; when OOMOS was sending, OOMOS was also receiving, etc. The same 2, 16, 40 and 120 byte packets were used (in TinyOS, only 16, 40 and 120 byte

packets were sent and received). Test application counted number of received packets from the total of 1000 packets transmitted, and calculated packet reception rate (PRR) for all four test modes. Test results are shown in Table 6.1.

Table 6.1: Packet reception rate (PRR) dependance on packet size

Packet size (bytes):	**2**	**16**	**40**	**120**
MansOS	99.7%	99.8%	100%	100%
OOMOS	99.9%	100%	100%	100%
Contiki	49.7%	95.4%	79.5%	59%
TinyOS	-	100%	100%	100%

It can be seen that Contiki, despite its fast transmission ability, is not able to receive many of transmitted packets. For small, 2 byte packets Contiki's PRR is under 50%, it is the highest for 16 byte packets (93%), and then falls again, for 120 byte packets Contiki's PRR is 59%. MansOS, OOMOS and TinyOS reception is very high, over 99.6% for all tested packet sizes.

TinyOS reception rate is stable at 100%, while MansOS and OOMOS are close to it: $\geq$ 99.7%. PRR of OOMOS is slightly higher than PRR of MansOS, that might be explained by slightly higher transmission rate of MansOS. If instead of relative PRR we take a look at absolute received packet count (Figure 6.7), it can be concluded, that MansOS reception performance is very close to Contiki, while OOMOS RX is slower. Actually OOMOS receives almost all transmitted packets, yet the transmission speed of the other OOMOS node is lower compared to MansOS and Contiki.

The author's conclusion: MansOS and OOMOS radio reception performance is very high and able to handle almost all the transmitted packets.

6.4.3 Optimizations

The author created initial OOMOS version without any optimizations to assess worst-case performance. As suggested in Section 6.4.1, OOMOS program binary image size is small enough to fit on TMote Sky sensor nodes, yet optimizations should be considered for OOMOS to be usable on more constrained platforms, and to be extensible with additional services in the future. In this section the author describes his experiment on OOMOS program size reduction, and its possible automation approaches.

One approach to code size reduction is dynamic exclusion of unused components, as it is successfully performed in MansOS. To test effectiveness of this approach, the author

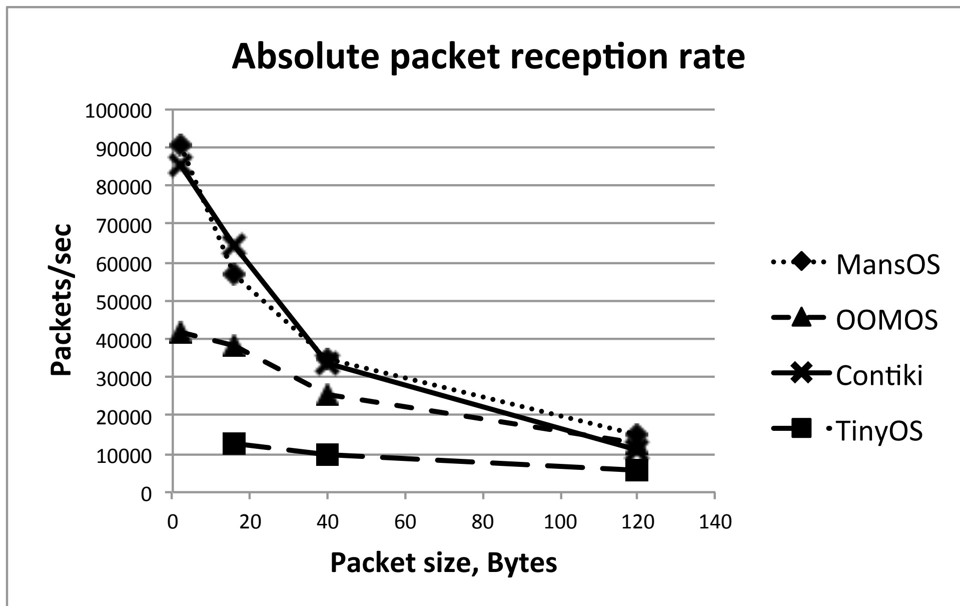

Figure 6.7: Absolute received packet count dependance on packet size - MansOS performance close to Contiki, OOMOS and TinyOS absolute reception lower due to lower performance of the transmitting node

performed manual exclusion of unused components for `RadioTxPerformance` application, where data packets are sent over radio. The unused interfaces and modules, that were commented out, are listed in Table 6.2. Visual result summary is shown in Figure 6.8.

Table 6.2: OOMOS code and RAM size optimization by excluding unused components

Excluded module	Flash saved, B	RAM saved, B
FastADC interface	154 (0.5%)	2
LEDs	1560 (5.2%)	54
LogStream interface	850 (2.8%)	0
External storage	1684 (5.6%)	16
Humidity sensor	3092 (10.2%)	8
Platform functions	268 (0.9%)	0
Total	7608 (25.2%)	80

Multiple unused interfaces and their implementations were removed: `IFastADC` (fast continuous sampling of a single ADC channel), `ILogStream` (ands its implementation in `IUART` and `IRadio` interfaces), `ILEDs`, `IStorage`. Also `GPIOLEDs` component was removed (generic LED control using GPIO pins), `M25P80` external flash memory driver, and `SHT11`

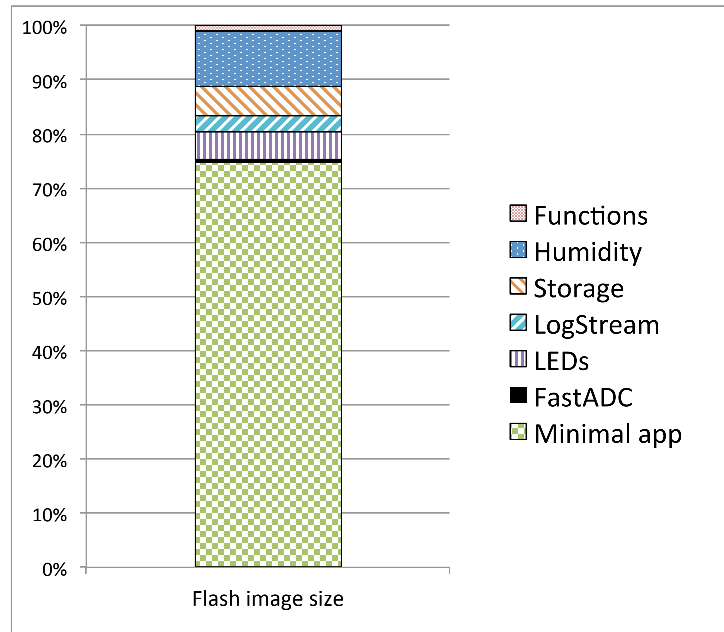

Figure 6.8: OOMOS code size reduction by excluding unused components - the author reduced program size by 25% (7608 bytes from initial 30206 bytes) by removing unused components in `RadioTxPerformance` application

humidity sensor driver.

Overall program size was reduced by 7608 bytes, i.e., 25% of initial size. It is a remarkable result. However, it must be noted, that this approach works only for small applications that use only a small subset of features provided by the OS. For an application using radio communication, debug stream over UART, sensors, external storage and LEDs, it would be difficult to exclude any components. Therefore this size reduction method allows to run constrained applications on more resource-constrained platforms.

The author performed component exclusion manually in this experiment. More research should be done on automated component exclusion. While MansOS uses automated approach by parsing the function call three and simply deciding which compiled object files to link together, a more complicated approach, using preprocessor may be needed in OOMOS, as there are features and components that are defined in multiple files, for example, interfaces and their implementers. Here the author lists suggestions on principles for further investigation:

- If a class is not used, its declaration and definition files (`.h` and `.cpp`) can be excluded.
- If an interface is not used, its declaration file can be excluded. Yet consequentially all the classes implementing it must be found and cleaned by removing inheritance

relation to this interface and removing all the functions implementing the interface.

- If some of class functions are never used, they can be removed.
- A multi-pass approach may be required to find unused classes. For example, `SHT11` humidity sensor driver may be used in `TelosB` platform class to provide humidity sensor functions declared in `IPlatform` class. When it is discovered, that application does not require humidity functionality, these functions can be removed and `SHT11` sensor class is no longer used.

6.4.4 Portability

According to *design rule#11*, portability to new hardware platforms is an important aspect for a WSN OS. The author performed source code statistical analysis to evaluate portability of proposed operating system prototypes. OS source code lines and files are counted and classified in the following categories:

- Platform-independent function interface,
- Platform-independent libraries and services,
- Kernel code, including task scheduling,
- Platform-independent device drivers,
- MSP430-architecture MCU code,
- AVR-architecture MCU code,
- Telosb platform-specific code,
- Zolertia Z1 platform-specific code,
- AVR-based (Arduino, AVR Raven, or Mica2) platform-specific code.

Such statistical analysis gives an overview of OS code size and distribution among categories, as well as platform-dependability and reusability of the source code.

The author compared source code of four operating systems: MansOS, OOMOS, Contiki OS and TinyOS. The analyzed operating systems support different feature sets and follow different ideologies, especially TinyOS. Therefore the following restrictions were followed to provide comparable results:

- Only meaningful source code lines were counted, without comments and blank lines.

- OOMOS is an experimental OS with the smallest implemented feature set of all the listed systems. Therefore only files implementing features available in OOMOS were counted for all four operating systems. For example, preemptive scheduling was excluded, and only physical layer communication was analyzed in the networking stack (inclusion of Contiki's sophisticated networking stack with 6lowpan IPv6 implementation would produce biased results).
- Although multiple other platforms are supported by MansOS, Contiki and TinyOS, TelosB and Zolertia Z1 were chosen as a baseline for portability evaluation. Zolertia Z1 has significant part common with TelosB, therefore large TelosB code base should be reusable for Z1.

Analyzed OS versions: Contiki 2.6, TinyOS 2.1, MansOS rev 452 (2012-09-11), OOMOS v1.0 (2012-08-31).

The author performed manual source file categorization, based on file location, name and content analysis. Lines of code for each file were counted, using `cloc` tool, v1.5 [168]. TinyOS nesC files were renamed to `.c` files and counted as C language code. The author empirically verified, that `cloc` tool correctly counts nesC lines of code in this manner.

Evaluation emphasizes the following aspects:

1. Total OS source code size comparison. This metric gives an overview of how much code is required for a working system.
2. Platform-independent code percentage, that describes code reusability.
3. Code size, that was required to implement TelosB platform. This aspect describes complexity of platform implementation without prior existing code.
4. Total code size for Zolertia Z1 platform, and percentage of code reused from existing TelosB platform.
5. Device driver source code size, describing complexity of new sensor, memory, radio chip and extension board adaptation.
6. Overall object-oriented OOMOS code overhead trend compared to procedural MansOS.

Total OS source code line count comparison is shown in Figure 6.9, total file count in Figure 6.10. Categorized source code statistics are depicted in Figure 6.11.

Figures show, that MansOS and OOMOS operating systems contain significantly less code, compared to Contiki and TinyOS. TinyOS code size is surprisingly huge: about 4 times the size of MansOS. The foundation lies in overly high modularity of TinyOS code

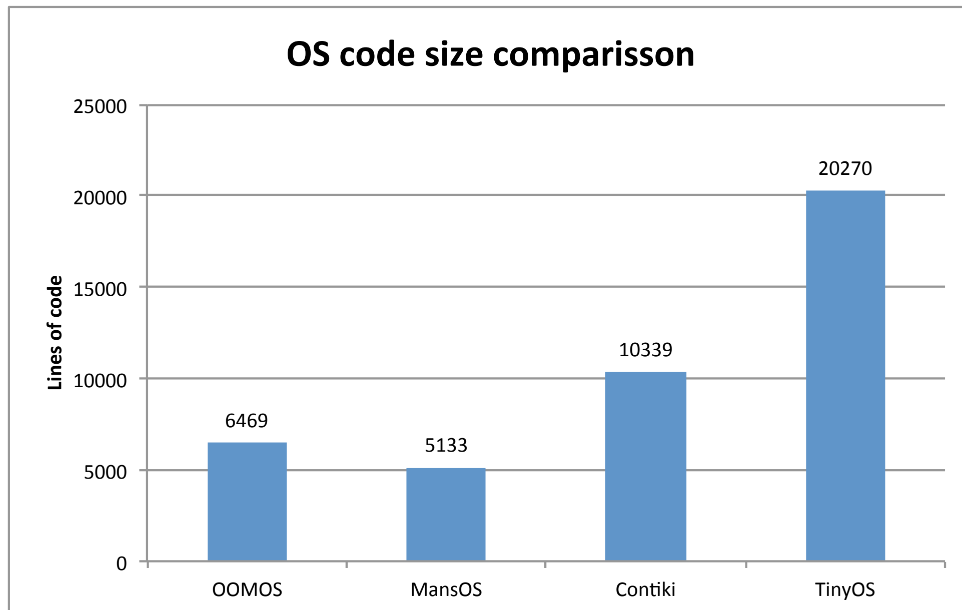

Figure 6.9: MansOS, OOMOS, Contiki and TinyOS size comparison, lines of source code - MansOS and OOMOS contain significantly less code, especially compared to TinyOS (only features available in all systems are included)

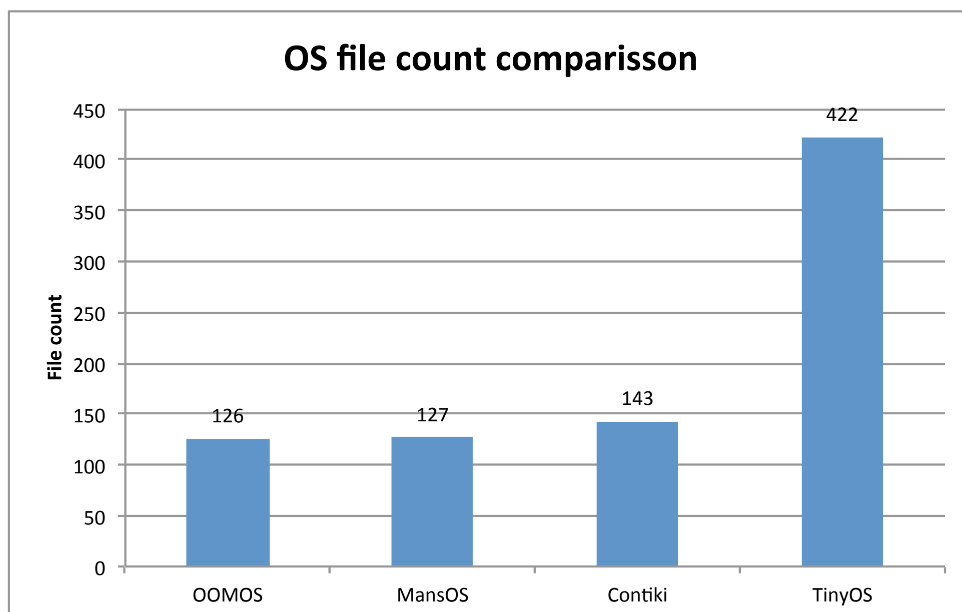

Figure 6.10: MansOS, OOMOS, Contiki and TinyOS size comparison, file count - high modularity of TinyOS system significant overhead for developers, having source code distributed across hundreds of files

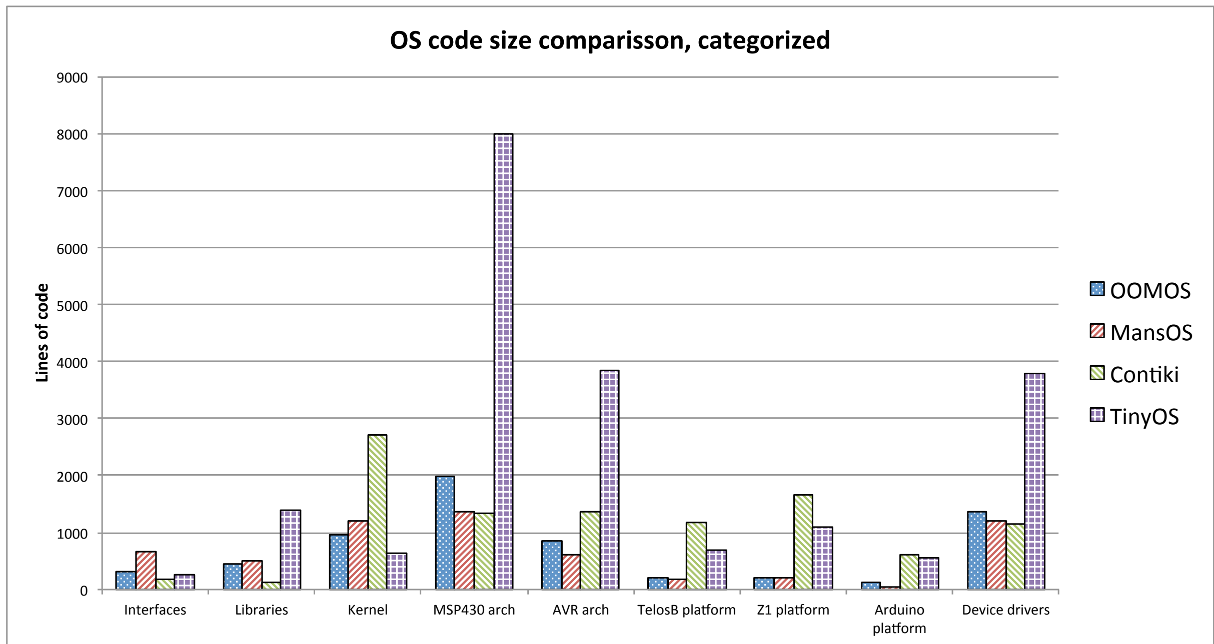

Figure 6.11: MansOS, OOMOS, Contiki and TinyOS code categorized

- the whole system is composed of hundreds of small components, implementing many interfaces and interconnected in complex structures. Although such modularity provides high degree of code reusability, such approach is very complex for both application and new platform developers. Programmers need to understand the whole structure and simultaneously be able to analyze each individual component, to decide, where a change of existing code or creation of new component is required.

Let us examine the whole OS source code, divided into two classes: platform-specific and platform-independent. Platform-dependent is the code, that initializes or describes a particular platform, and is not directly reusable for new platform adaptation. In the particular case, TelosB, Zolertia Z1 and AVR platform-specific initialization code sum is described as platform-specific, while the rest is platform-independent. Source code line count is shown in Figure 6.12.

Overall, MansOS and OOMOS contain more reusable code: 91.8% and 91.6% respectively, while TinyOS is less portable (88% reusable), and Contiki contains most platform-specific code (only 67% reusable). It is due the fact, that both TelosB and Z1 platform code in Contiki contains parts, that are equally usable for multiple platforms, for example, analog sensor support is duplicated in both platforms.

To get a perception of code size and effort required to adapt a new platform, let us examine Zolertia Z1 platform in all four operating systems. Z1 platform source code line count is shown in Figure 6.13.

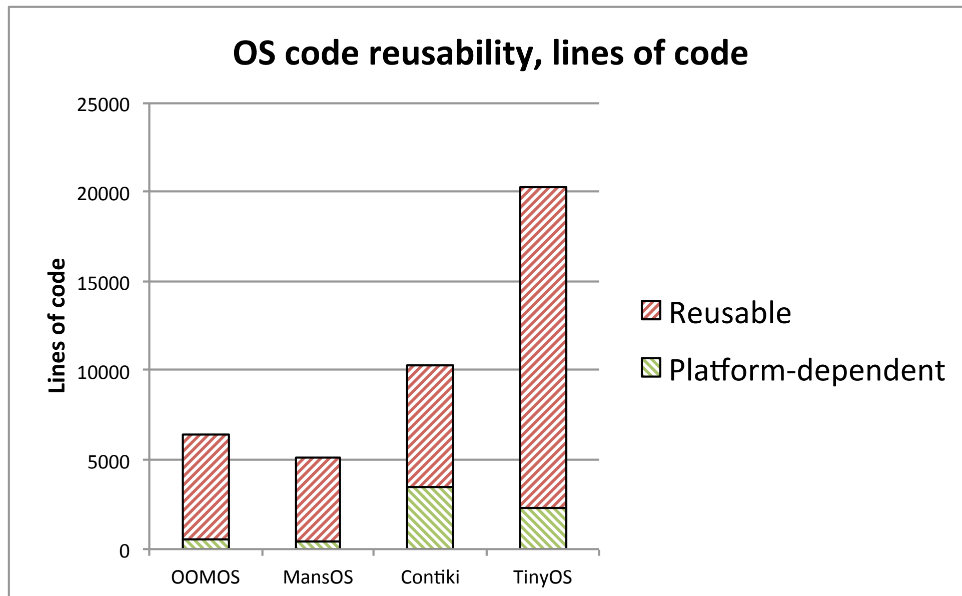

Figure 6.12: OOMOS, MansOS, Contiki and TinyOS reusability, lines of code - MansOS and OOMOS contain few lines of non-reusable code, 8.2% and 8.4% respectively, while TinyOS contains 12% and Contiki - 33% of platform-dependent, non-reusable source code

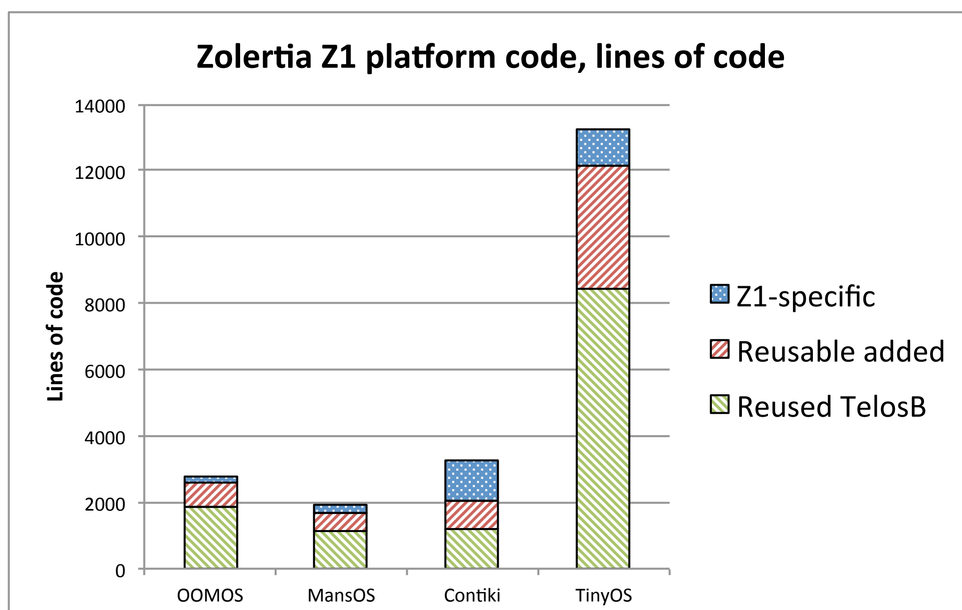

Figure 6.13: Zolertia Z1 platform in OOMOS, MansOS, Contiki and TinyOS, lines of code - MansOS and OOMOS require few lines of additional code, and significant part of added code is further reusable

Figures show, that MansOS and OOMOS porting to Zolertia Z1 platform required writing of significantly less code, compared to Contiki: 59% of TelosB code in MansOS was reused, 66% in OOMOS, 36% in Contiki. In addition, newly created Z1 platform code is further reusable for other platform adaptation: 89% of added code is further reusable in MansOS, 93% in OOMOS. In Contiki code is mostly platform-specific, without a goal to reuse it later: only 63% of added code is reusable.

When comparing TinyOS to OOMOS, TinyOS results are comparable in relative numbers and even slightly better compared to MansOS: 64% of TelosB code is reusable in TinyOS and 92% of newly added Zolertia Z1 code is further reusable. Yet, as mentioned above, in absolute numbers TinyOS code base is huge: 4.7 times larger than OOMOS and 6.9 times larger than MansOS (source code lines, see Table 6.3), requiring much more effort for platform porting.

Table 6.3: Zolertia Z1 platform source code size in OOMOS, MansOS, Contiki and TinyOS

Code category	OOMOS	MansOS	Contiki	TinyOS
Reused TelosB	1847	1126	1178	8458
Reusable added	751	579	852	3666
Z1-specific	206	203	1206	1103
Total	2804	1908	3236	13227

Device driver development requires a lot of effort in new platform adaptation process. Therefore let us examine device driver code size comparison. Lines of code are shown in Figure 6.14, file count - Figure 6.15.

It can be seen, that OOMOS driver development requires slightly more coding effort compared to MansOS. The reason is object-oriented nature of the drivers - each devices is represented by a class, requiring both declaration and definition. In general, class oriented code contains more code lines compared to procedural approach. It must be noted, that all device drivers in MansOS and OOMOS are written using platform-independent routines and the code is therefore reusable for multiple platforms.

Contiki contains less platform-independent device driver code, compared to MansOS and OOMOS. However, additional device drivers are platform-specific and included in platform code. When considering also these platform-specific drivers (in particular, for ADXL345 accelerometer, TMP102 temperature sensor and M25Px flash memory chips), the driver code is larger than MansOS and OOMOS. Transforming these drivers to platform-independent code would increase portability of Contiki.

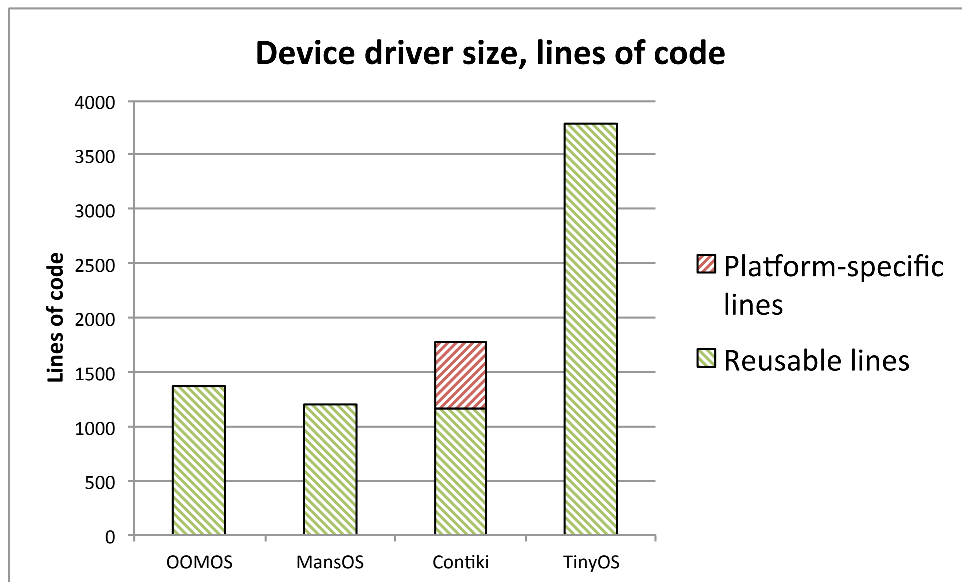

Figure 6.14: Device driver code in OOMOS, MansOS, Contiki and TinyOS, lines of code - OOMOS device drivers contain slightly more code compared to MansOS, yet less than Contiki and TinyOS

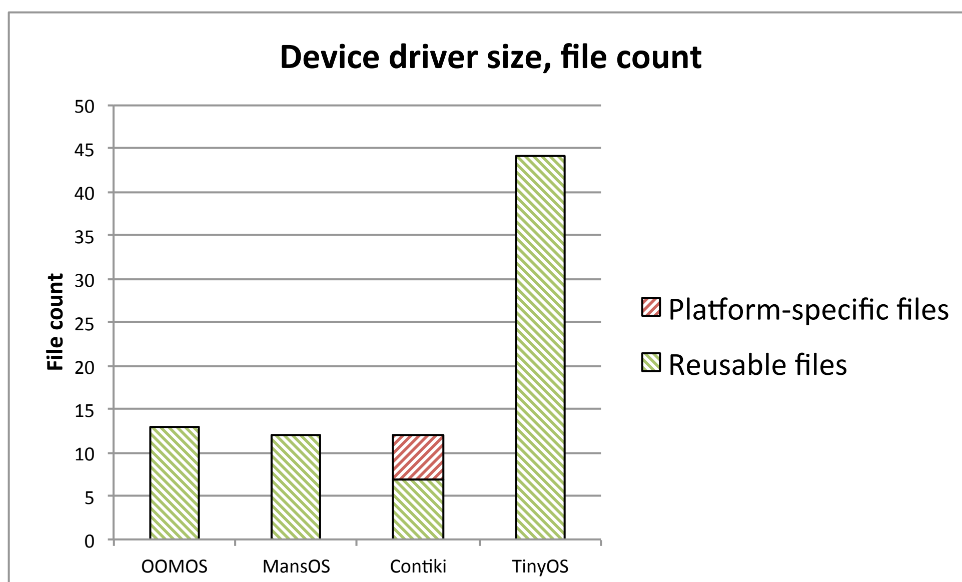

Figure 6.15: Device driver file count in OOMOS, MansOS, Contiki and TinyOS

TinyOS device drivers contain 3 times more source code lines compared to MansOS (3791 and 1205 lines respectively), and almost 4 times more files (44 versus 12). Driver development in TinyOS is therefore significantly more complicated.

To summarize, MansOS and OOMOS were developed with portability (*design rule#11*) in mind right from the beginning. Results show that conformance to this rule provides high portability in terms of source code reusability.

6.4.5 Object-orientation overhead

This section compares MansOS and OOMOS to understand impact to code size of introducing object-orientation, while maintaining the same overall OS ideology and structure. OOMOS and MansOS source line count is compared in Figure 6.16, file count comparison is depicted in Figure 6.17.

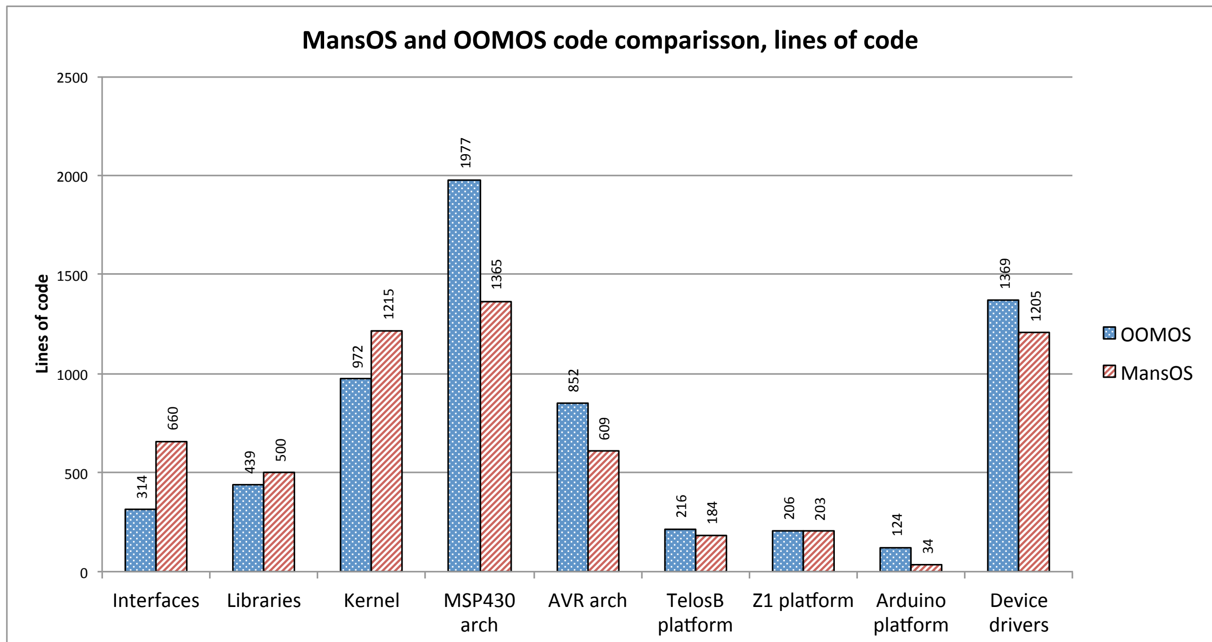

Figure 6.16: Object-oriented OOMOS compared to procedural MansOS, lines of code

These graphs show the ability of object-oriented approach to better separate interface and implementation code. It can be seen, that OOMOS interface files contain less code lines. OOMOS contains more empty interface declarations, that are implemented in the library code, while in MansOS part of interfaces are implemented straight in the interface code.

Kernel size is comparable for both systems. MansOS kernel is even slightly larger, as it performs generic initialization for all platforms, while OOMOS delegates initialization

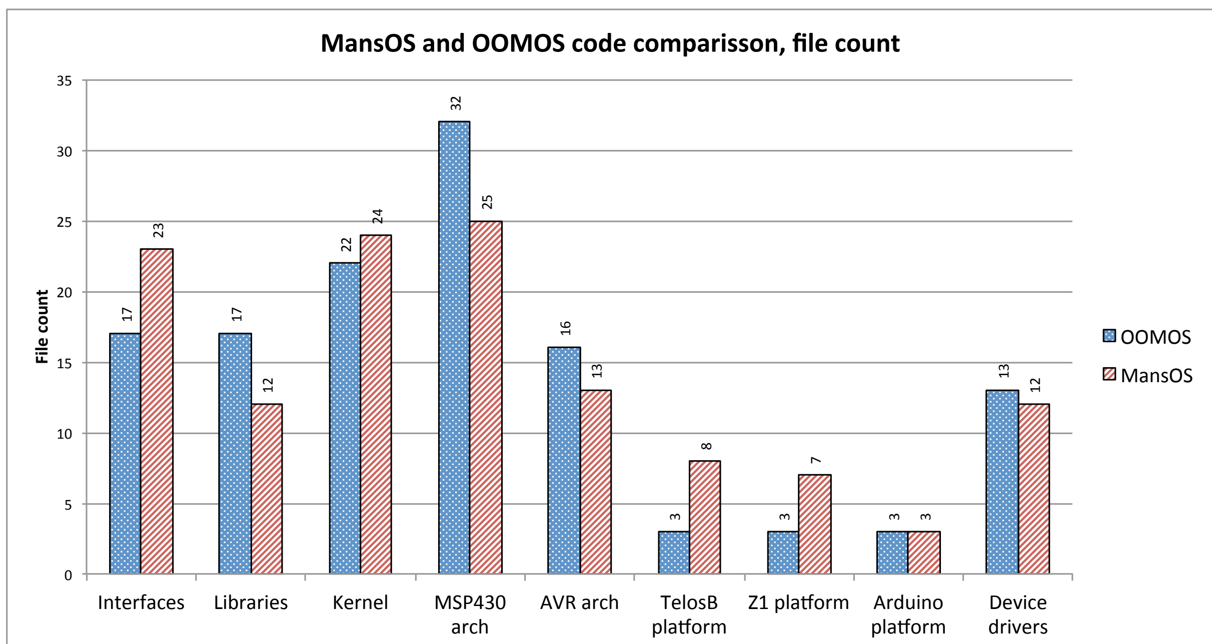

Figure 6.17: Object-oriented OOMOS compared to procedural MansOS, file count

process to each individual platform code.

MSP430 architecture code shows, that object-oriented approach requires more lines of code for MCU support. Partially it is due to higher modularity - MCU modules (such as `MSP430_TimerA3` and `MSP430_USART1`) are extended to separated classes, while MansOS uses lower degree of modularity.

One feature of OOMOS is not visible in the graphs - OOMOS contains separate classes for each particular MCU model. For example, MSP430F1611 and MSP430F2617. These classes include all the required MCU modules and are an effective approach to reusability - whenever a new platform contains a previously defined MCU, it simply has to include the particular class in its platform code, without requirement to combine and interconnect all the MCU modules separately for each platform.

As already mentioned above, device driver development in object-oriented fashion produces more source code, and for the sake of higher modularity, code is more fragmented across files. However, such approach provides higher module reusability.

6.5 Future work according to design rules

OOMOS is an experimental WSN operating system, designed by the author incrementally according to rules. Although it can be used for simple applications already at this stage, significant future work is required to implement features necessary for wide application

range. The main challenges involve implementation of a full networking protocol stack, preemptive scheduling and important services, including file system, time synchronization and remote access. This section demonstrates how the proposed design rules can be used as a useful tool for design and implementation of these features.

6.5.1 Networking protocol stack

At the moment of writing this thesis the author had implemented only physical layer (PHY) protocol responsible for sending packets (terms *layer* and *protocol* are used interchangeably in this section). Basic CSMA MAC mechanisms are built in 802.15.4-based radio chips, such as CC2420. But they don't provide full flexibility that might be required by custom user needs. Implementation of flexible networking stack is an important task. This section describes how a flexible protocol stack can be designed. The design rules proposed in this thesis help in identifying important requirements and choosing between alternative approaches for the protocol stack.

Implementation of the stack consists of two parts. First, a framework or interface must be established supporting implementation of different protocol stacks. Second, particular protocols are implemented using the framework available.

The following requirements for the networking protocol framework can be inferred from the proposed design rules:

- Networking stack should be flexible with ability to change each layer separately. Users should be able to design their own protocols easily and modify the stack by replacing individual protocol implementations (*design rule#6*).
- Each protocol must be able to track the state of the node and the whole network and decide on further actions. CSMA protocols (*design rule#5*) might require counters to track number of transmission retries, TDMA protocols might require individual timer.
- Implementation of IPv6 protocol stack should be possible (*design rule#8*). It may be difficult to implement the whole stack in the OOMOS framework, wrapper layer of existing IPv6 libraries could be used.

The following rules are targeted to individual protocol development (not the general protocol stack framework):

- The default protocols should be sink-oriented (*design rule#1*) and consider powered motes in the network (*design rule#2*). I.e., it should be possible to notify each networking protocol whether this node has unlimited energy resources and can operate

in different mode. The platform class should have a function that signals whether this particular sensor node has access to extended power source and can therefore run on a higher duty-cycle.

- Acknowledgement mechanism for reliable delivery should be included in one of the protocols (*design rule#7*). It can be implemented either as a separate layer, or as a part of other layers.

There are several ways how to implement network protocols. One approach is to create a separate thread for each layer. However, the author does not recommend it, as it would make protocols more dependent on the selected scheduling technique. The same protocol stack implementation should be available for multiple scheduling techniques (*design rule#16, 17, and 18*). Another observation: protocols should be reusable and interchangeable - the same interface should be provided and used by each protocol. Definition of connection between protocols is necessary.

In summary, the author suggests the following networking stack framework:

- Each layer (or protocol) is represented by an abstract base class `AbstractProtocol` that implements part of `IProtocol` interface. See prototype in Listing D.7.

- Each protocol holds pointers for protocols above and below it and functions to modify these relations. Initializing, removing or replacing protocols is simple in such architecture.

- Each protocol has functions `pushUp()` and `pullDown()` that transfer data to the next layer up/down accordingly.

- Default implementations of application layer and physical layer protocols is provided by the OS. These act as wrappers between methods used in the protocol stack and methods used by application and radio chip driver. It is required to hide implementation details from users who may not understand the protocol stack idea.

- Each protocol can use one timer for delayed operations.

- Due to resource efficiency reasons sharing a single buffer between all protocols is suggested without duplication. A mechanism how to calculate size of header information for each protocol must be ensured. A separate class for handling the packet buffer issues is suggested.

6.5.2 Services and scheduling

In addition to networking protocol stack implementation OOMOS can be improved by implementing more services. Design rules are important in this aspect as a tool for suggesting which features should be considered first.

Preemptive scheduling is one of the features that might support requirements for wide application range (*design rule#17*). This feature requires highly efficient implementation and integration into kernel. Therefore implementation of this feature is up to OS developers.

In contrast, other features, suggested by design rules, can be implemented as external services or libraries: file system (*design rule#20*), time synchronization (*design rule#21*) and remote access (*design rule#25*). Contiki OS is an example of file system implementation (Coffee) that is independent from the kernel and can be ported even to different operating systems [169]. Time synchronization might require modifications in the operating system if very high accuracy is required [122]. However, if accuracy in millisecond range is acceptable, time synchronization can be implemented either as an option in the networking protocols or as a separate service. Support from networking protocols is needed in any case, as time synchronization protocols should be able to transmit data exactly at the moments when it is necessary. Remote reprogramming can be part of the OS to be very optimized (as it is in the case of MansOS) or it can also be implemented as an additional service that uses low-level program memory access [170].

6.6 Summary

In this chapter author presented OOMOS - an object-oriented OS that he implemented according to proposed design rules. It is a work in progress and space for improvement was identified in this chapter. As the evaluation showed, OOMOS achieves high portability and reasonable performance. This chapter demonstrated proposed design rules as an important tool already in early stages of WSN software development: during design and implementation of WSN operating systems.

7 Conclusion

Development of software for wireless sensor networks is a complex task due to multiple reasons. Resource constrained devices are combined with unreliable communication channels to form complex distributed systems. Hardware platforms are either customized or fully custom built for particular applications. A reusable and extensible source code base is required to simplify the task of WSN programming. Central thesis of this work states that wireless sensor network software development requires a methodology as an important step towards standardization.

The author analyzed a set of 40 WSN deployments described in scientific literature in order to develop the said methodology. In addition, he has also participated in multiple WSN research projects [16, 18, 19, 20, 21, 22, 25, 26, 27]. He identified critical WSN properties based on the deployment survey and WSN project experience. He then proposed a set of design rules addressing these problems forming a methodology for WSN software development. The author performed qualitative evaluation of the rules by applying them to different WSN software entities. The evaluation consisted of identifying the problems that each WSN software entity (deployment or operating system) contains and analysis of solutions, based on the design rules. Results showed that proposed design rules serve as a valuable methodology at different stages of software development: design and planning, programming and testing, as well as assessment and improvement analysis.

The author's main contribution in this thesis includes:

1. Analysis of 40 sensor network deployments described in the research literature. As a result the critical and recurring WSN properties were distilled.

2. Identification of common WSN design problems that identify the challenges based on critical WSN properties and user requirements.

3. Introduction of a WSN software development methodology in the form of 25 design rules and analysis of their mapping to underlying problems.

4. Evaluation of the proposed design rule impact on existing WSN software improvement. Design rules are shown as a tool for existing system comparison, drawback identification and future direction sketch. The evaluation consists of three parts:

   (a) Improvements to the analyzed deployment set showing design rule applicability in general, for WSN users.

   (b) Existing operating system conformance to proposed rules and suggestions for OS improvement. Design rules are shown as an important tool for WSN OS developers. This evaluation includes the author's participation in the development and improvement analysis of MansOS: a portable operating system (OS) for sensor networks.

   (c) A wearable sensor network use-case scenario - assessment of prototype implementation and suggestions for future work. This part shows more detailed improvement of a particular WSN deployment in terms of network lifetime and network coverage.

5. In addition, the author has developed a new WSN operating system, (called Object-Oriented MansOS or OOMOS) according to the rules. This part of the thesis shows design rules as a valuable tool in early stages of WSN OS design and implementation.

It is complicated in practice to measure external deployments in quantitative terms. For example, it is relatively simple to calculate energy consumption for our own sensor networks. However, it is very complicated to estimate energy efficiency of an external WSN deployment that is neither physically accessible to the analyst, nor is described in rich detail by the deployment designers. The author of this thesis extracted technical WSN deployment characteristics based on the information available. In some cases, including design rule evaluation, it is not possible to gather quantitative information. Qualitative discussion was used in such situations. Nevertheless, to the best of the authors knowledge, this work proposes the most comprehensive and formalized set of design rules for WSN software development.

The gaps identified during thesis research work serve as starting points for future research directions:

- WSN operating systems are only one possible software abstraction. Further research on WSN middleware on top of MansOS and other operating systems can be explored.

- Sensor networks is an evolving field and WSN deployment survey should be updated periodically to follow the state-of-the-art and recent trends.

- Deployment analysis reveals that energy harvesting is a rarely explored direction. The author believes, it has a great potential for future research.
- OOMOS is usable for WSN applications with typical tasks. Yet it still is a prototype covering only small set of essential OS parts. It should be further extended to a feature rich OS to understand scalability and trends of object-oriented operating systems for WSNs.
- While OOMOS provides reasonable performance there is still place for optimizations.

The design rules proposed in this thesis represent an important step towards higher wireless sensor network standardization and software portability. As such, this thesis is an important milestone for sensor network maturity and wide distribution.

References

[1] Pei Zhang, Christopher M. Sadler, Stephen A. Lyon, and Margaret Martonosi. **Hardware design experiences in ZebraNet**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 227–238, New York, NY, USA, 2004. ACM. Available from: http://doi.acm.org/10.1145/1031495.1031522. 13

[2] L. Selavo, A. Wood, Q. Cao, T. Sookoor, H. Liu, A. Srinivasan, Y. Wu, W. Kang, J. Stankovic, D. Young, and J. Porter. **LUSTER: wireless sensor network for environmental research**. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 103–116, New York, NY, USA, 2007. ACM. Available from: http://doi.acm.org/10.1145/1322263.1322274. 13

[3] Geoff Werner-Allen, Konrad Lorincz, Jeff Johnson, Jonathan Lees, and Matt Welsh. **Fidelity and yield in a volcano monitoring sensor network**. In *Proceedings of the 7th symposium on Operating systems design and implementation*, OSDI '06, pages 381–396, Berkeley, CA, USA, 2006. USENIX Association. Available from: http://portal.acm.org/citation.cfm?id=1298455.1298491. 13, 34, 98

[4] Ryan Aylward and Joseph A. Paradiso. **A compact, high-speed, wearable sensor network for biomotion capture and interactive media**. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 380–389, New York, NY, USA, 2007. ACM. Available from: http://doi.acm.org/10.1145/1236360.1236408. 13, 34

[5] Georg Wittenburg, Kirsten Terfloth, Freddy López Villafuerte, Tomasz Naumowicz, Hartmut Ritter, and Jochen Schiller. **Fence monitoring: experimental evaluation of a use case for wireless sensor networks**. In *Proceedings of the 4th European conference on Wireless sensor networks*, EWSN'07, pages 163–178, Berlin, Heidelberg, 2007. Springer-Verlag. Available from: http://portal.acm.org/citation.cfm?id=1758126.1758141. 13, 35

[6] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. **A line in the sand: a wireless sensor network for target detection, classification, and tracking**. *Computer Networks*, **46**(5):605 – 634, 2004. Military Communications Systems and Technologies. Available from: http://www.sciencedirect.com/science/article/pii/S138912860400146X. 13, 33

[7] Virantha Ekanayake, Clinton Kelly, IV, and Rajit Manohar. **An ultra low-power processor for sensor networks**. In *Proceedings of the 11th international conference on Architectural support for programming languages and operating systems*, ASPLOS-XI, pages 27–36, New York, NY, USA, 2004. ACM. Available from: http://doi.acm.org/10.1145/1024393.1024397. 14

[8] Texas Instruments. **CC2420: Single-Chip 2.4 GHz IEEE 802.15.4 Compliant and ZigBee Ready RF Transceiver**. Available from: http://www.ti.com/lit/gpn/cc2420 [cited 2011.07.13]. 14, 41, 57, 97

[9] Ian F. Akyildiz, Dario Pompili, and Tommaso Melodia. **Underwater acoustic sensor networks: research challenges**. *Ad Hoc Networks*, **3**(3):257 – 279, 2005. Available from: http://www.sciencedirect.com/science/article/pii/S1570870505000168. 14

[10] Gabe Cohn, Erich Stuntebeck, Jagdish Pandey, Brian Otis, Gregory D. Abowd, and Shwetak N. Patel. **SNUPI: sensor nodes utilizing powerline infrastructure**. In *Proceedings of the 12th ACM international*

conference on Ubiquitous computing, Ubicomp '10, pages 159–168, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1864349.1864377`. 14

[11] Karlis Prieditis, Ivars Drikis, and Leo Selavo. **SAntArray: passive element array antenna for wireless sensor networks**. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 433–434, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1869983.1870060`. 14, 43

[12] **IEEE 802.15™: WIRELESS PERSONAL AREA NETWORKS (PANs)**. Available from: `http://standards.ieee.org/about/get/802/802.15.html` [cited 2011.07.13]. 14

[13] I. Demirkol, C. Ersoy, and F. Alagoz. **MAC protocols for wireless sensor networks: a survey**. *Communications Magazine, IEEE*, **44**(4):115–121, april 2006. 14

[14] Brad Karp and H. T. Kung. **GPSR: greedy perimeter stateless routing for wireless networks**. In *Proceedings of the 6th annual international conference on Mobile computing and networking*, MobiCom '00, pages 243–254, New York, NY, USA, 2000. ACM. Available from: `http://doi.acm.org/10.1145/345910.345953`. 14

[15] A. Dunkels, T. Voigt, and J. Alonso. **Making TCP/IP viable for wireless sensor networks**. In *Proceedings of the First European Workshop on Wireless Sensor Networks (EWSN 2004), work-in-progress session, Berlin, Germany*, 2004. 14

[16] Atis Elsts, Girts Strazdins, Andrey Vihrov, and Leo Selavo. **Design and Implementation of MansOS: a Wireless Sensor Network Operating System**. In *Scientific Papers*. University of Latvia, 2012. 14, 23, 24, 25, 45, 116

[17] Girts Strazdins, Atis Elsts, and Leo Selavo. **MansOS: Easy to Use, Portable and Resource Efficient Operating System For Networked Embedded Devices**. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 427–428, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1869983.1870057`. 14, 24

[18] Reinholds Zviedris, Atis Elsts, Girts Strazdins, Artis Mednis, and Leo Selavo. **Lynxnet: Wild animal monitoring using sensor networks**. In Peter Corke Pedro J. Marron, Thiemo Voigt and Luca Mottola, editors, *Real-World Wireless Sensor Networks 4th International Workshop, REALWSN 2010, Colombo, Sri Lanka, December 16-17, 2010. Proceedings*, **6511** of *Lecture Notes in Computer Science*, pages 170–173. Springer-Verlag GmbH, 2010. 14, 116

[19] Atis Elsts, Rihards Balass, Janis Judvaitis, Reinholds Zviedris, Girts Strazdins, Artis Mednis, and Leo Selavo. **SADmote: A Robust and Cost-Effective Device for Environmental Monitoring**. In Andreas Herkersdorf, Kay Romer, and Uwe Brinkschulte, editors, *Architecture of Computing Systems – ARCS 2012*, **7179** of *Lecture Notes in Computer Science*, pages 225–237. Springer Berlin / Heidelberg, 2012. Available from: `http://dx.doi.org/10.1007/978-3-642-28293-5_19`. 14, 116

[20] Rinalds Ruskuls, Girts Strazdins, and Leo Selavo. **Accurate Sensor Node Energy Consumption Estimation Using EdiMote Prototyping Platform**. In *In the 3rd InternationalWorkshop on Networks of Cooperating Objects (CONET'12), Electronic Proceedings of CPSWeek'12*, 2012. 14, 116

[21] Girts Strazdins. **Location Based Information Storage and Dissemination in Vehicular Ad Hoc Networks**. In Janis Grundspenkis, Marite Kirikova, Yannis Manolopoulos, and Leonids Novickis, editors, *Advances in Databases and Information Systems Associated Workshops and Doctoral Consortium of the 13th East European Conference, ADBIS 2009, Riga, Latvia, September 7-10, 2009. Revised Selected Papers*, **5968** of *Lecture Notes in Computer Science*, pages 211–219. Springer Berlin / Heidelberg, 2010. 10.1007/978-3-642-12082-427. Available from: `http://dx.doi.org/10.1007/978-3-642-12082-4_27`. 14, 116

[22] Artis Mednis, Girts Strazdins, Martins Liepins, Andris Gordjusins, and Leo Selavo. **RoadMic: Road Surface Monitoring Using Vehicular Sensor Networks with Microphones**. In *Proc. of Networked Digital Technologies, Part II: Second International Conference, NDT 2010*, pages 417–429. Springer-Verlag GmbH, 2010. Available from: `http://www.springerlink.com/content/q3t5564544t8x188/fulltext.pdf`. 14, 15, 116

[23] Girts Strazdins, Artis Mednis, Georgijs Kanonirs, Reinholds Zviedris, and Leo Selavo. **Towards Vehicular Sensor Networks with Android Smartphones for Road Surface Monitoring**. In *Proc. of the 2nd International Workshop on Networks of Cooperating Objects (CONET'11), Electronic Proceedings of CPSWeek'11*, page 4, April 2011. 14

[24] Artis Mednis, Girts Strazdins, Reinholds Zviedris, Georgijs Kanonirs, and Leo Selavo. **Real time pothole detection using Android smartphones with accelerometers**. In *Distributed Computing in Sensor Systems and Workshops (DCOSS), 2011 International Conference on*, pages 1–6, june 2011. Available from: `http://ieeexplore.ieee.org/search/freesrchabstract.jsp?tp=&arnumber=5982206`. 14

[25] Girts Strazdins, Artis Mednis, Reinholds Zviedris, Georgijs Kanonirs, and Leo Selavo. **Virtual Ground Truth in Vehicular Sensing Experiments: How to Mark it Accurately**. In *The 5th International Conference on Sensor Technologies and Applications (SENSORCOMM 2011)*, Nice, France, August 2011. 14, 116

[26] Nikolajs Agafonovs, Girts Strazdins, and Modris Greitans. **Accessible, Customizable, High-Performance IEEE 802.11p Vehicular Communication Solution**. In *Proc. of 1st International Workshop on Vehicular Communications and Applications (VCA 2012)*, pages 127–132, June 2012. 14, 116

[27] Girts Strazdins, Andris Gordjusins, Georgijs Kanonirs, Vadims Kurmis, Artis Mednis, Reinholds Zviedris, and Leo Selavo. **Team "Latvia" GCDC 2011 Technical Paper**. Technical report, Institute of Electronics and Computer Science (EDI) and University of Latvia, April 2011. 14, 116

[28] Nikolajs Agafonovs, Andrejs Skageris, Girts Strazdins, and Artis Mednis. **IMilePost: Embedded Solution for Dangerous Road Situation Warnings**. In *Prof of the 1st IEEE/ASME International Conference on Artificial Intelligence, Modelling and Simulation, AIMS2013*, page 6, 2013. 14

[29] Girts Strazdins, Sashidharan Komandur, and Arne Styve. **Kinect-based Systems For Maritime Operation Simulators?** In *27th European Conference on Modelling and Simulation (ECMS'13)*, pages 205–211, May 2013. 14

[30] Girts Strazdins, Atis Elsts, Krisjanis Nesenbergs, and Leo Selavo. **Wireless Sensor Network Operating System Design Rules Based on Real-World Deployment Survey**. *Journal of Sensor and Actuator Networks*, **2**(3):509–556, 2013. Available from: `http://www.mdpi.com/2224-2708/2/3/509`. 14, 52

[31] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M.B. Srivastava. **Participatory Sensing**. In *Proc. of World Sensor Web Workshop (WSW'06), collocated with SenSys'06*, pages 1–5, 2006. 15

[32] **TMote Sky datasheet**. Available from: `http://www.eecs.harvard.edu/~konrad/projects/shimmer/references/tmote-sky-datasheet.pdf` [cited 2011-07-13]. 15, 41, 97

[33] Aman Kansal, Jason Hsu, Sadaf Zahedi, and Mani B. Srivastava. **Power management in energy harvesting sensor networks**. *ACM Transactions on Embedded Computing Systems, Special Section LCTES'05*, **6**, September 2007. Available from: `http://doi.acm.org/10.1145/1274858.1274870`. 15

[34] **IEEE 802.15 WPAN Task Group 4 (TG4)**. Available from: `http://www.ieee802.org/15/pub/TG4.html` [cited 26.07.2011.]. 16, 41, 57

[35] Anna Hac. *Wireless sensor Network Designs*. John Wiley and Sons, Ltd, 2003. 18

[36] Jason Hill, Mike Horton, Ralph Kling, and Lakshman Krishnamurthy. **The platforms enabling wireless sensor networks**. *Commun. ACM*, **47**:41–46, June 2004. Available from: `http://doi.acm.org/10.1145/990680.990705`. 18

[37] Sameer Tilak, Nael B Abu-Ghazaleh, and Wendi Heinzelman. **A taxonomy of wireless micro-sensor network models**. *ACM SIGMOBILE Mobile Computing and Communications Review*, **6**(2):28–36, 2002. 18

[38] Luca Mottola and Gian Pietro Picco. **Programming wireless sensor networks: Fundamental concepts and state of the art**. *ACM Computing Surveys*, **43**:19:1–19:51, April 2011. Available from: `http://doi.acm.org/10.1145/1922649.1922656`. 18

[39] Jan Beutel. **Metrics for Sensor Network Platforms**. In *Proc. ACM Workshop on Real-World Wireless Sensor Networks (REALWSN'06)*, page 5, 2006. 18

[40] K. Romer and F. Mattern. **The design space of wireless sensor networks**. *Wireless Communications, IEEE*, **11**(6):54–61, 2004. 18

[41] Vlado Handziski, Andreas Kopke, Holger Karl, and Adam Wolisz. **A common wireless sensor network architecture?** Technical report, Telecommunications Networks Group, Technische Universitat Berlin, 2003. 18

[42] Jason Lester Hill. *System architecture for wireless sensor networks*. PhD thesis, University of California, 2003. 18

[43] Vivek Mhatre and Catherine Rosenberg. **Design guidelines for wireless sensor networks: communication, clustering and aggregation**. *Ad Hoc Networks*, **2**(1):45–63, 2004. 18

[44] S. Olariu and I. Stojmenovic. **Design Guidelines for Maximizing Lifetime and Avoiding Energy Holes in Sensor Networks with Uniform Distribution and Uniform Reporting**. In *INFOCOM 2006. 25th IEEE International Conference on Computer Communications. Proceedings*, pages 1–12, April 2006. 18

[45] Ivan Stojmenovic, Amiya Nayak, and Johnson Kuruvila. **Design guidelines for routing protocols in ad hoc and sensor networks with a realistic physical layer**. *Communications Magazine, IEEE*, **43**(3):101–106, 2005. 18

[46] Felix Jonathan Oppermann and Steffen Peter. **Inferring technical constraints of a wireless sensor network application from end-user requirements**. In *Mobile Ad-hoc and Sensor Networks (MSN), 2010 Sixth International Conference on*, pages 169–175. IEEE, 2010. 19

[47] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. **Contiki - A Lightweight and Flexible Operating System for Tiny Networked Sensors**. In *Proc. of Annual IEEE Conference on Local Computer Networks*, pages 455–462, Los Alamitos, CA, USA, 2004. IEEE Computer Society. 23, 28, 45, 57, 90

[48] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al. **Tinyos: An operating system for sensor networks**. *Ambient intelligence*, **35**, 2005. 23, 27, 45, 57, 88

[49] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian 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*, IPSN '08, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society. Available from: `http://dx.doi.org/10.1109/IPSN.2008.54`. 23, 29, 42, 45

[50] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. **MANTIS OS: An embedded multithreaded operating system for wireless micro sensor platforms**. *Mobile Networks and Applications*, **10**(4):563–579, 2005. 23, 29

[51] Yu-Ting Chen, Ting-Chou Chien, and Pai H. Chou. **Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms**. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 183–196, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1869983.1870002`. 23

[52] Hubert Zimmermann. **OSI reference model–The ISO model of architecture for open systems interconnection**. *Communications, IEEE Transactions on*, **28**(4):425–432, 1980. 27, 162

[53] K. Klues, C.J.M. Liang, J. Paek, R. Musăloiu-e, P. Levis, A. Terzis, and R. Govindan. **TOSThreads: Thread-Safe and Non-invasive Preemption in TinyOS**. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09)*, pages 127–140. ACM, 2009. 28

[54] Contiki Developers. **Mailing List Discussion Archive**. Available from: `http://sourceforge.net/mailarchive/forum.php?forum_name=contiki-developers` [cited 2014-02-01]. 28

[55] **Arduino**. Available from: `http://arduino.cc/` [cited 2014-02-01]. 29

[56] RASPBERRY PI FOUNDATION. **Raspberry Pi** [online]. Available from: http://www.raspberrypi.org/ [cited 2014-01-28]. 30, 49

[57] HARDKERNEL. **Odroid** [online]. Available from: http://www.hardkernel.com/ [cited 2014-01-28]. 30, 49

[58] J. ELLUL, B. LO, AND G.Z. YANG. **The BSNOS Platform: A Body Sensor Networks Targeted Operating System and Toolset**. In *SENSORCOMM 2011, The Fifth International Conference on Sensor Technologies and Applications*, pages 381–386, 2011. 30, 87

[59] N. BROUWERS, K. LANGENDOEN, AND P. CORKE. **Darjeeling, A Feature-Rich VM for the Resource Poor**. In *7th ACM Conf. on Embedded Networked Sensor Systems (SenSys'09)*, pages 169–182, Berkeley, CA, November 2009. 30

[60] FAISAL ASLAM, CHRISTIAN SCHINDELHAUER, GIDON ERNST, DAMIAN SPYRA, JAN MEYER, AND MOHANNAD ZALLOOM. **Introducing TakaTuka: a Java virtualmachine for motes**. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 399–400, New York, NY, USA, 2008. ACM. Available from: http://doi.acm.org/10.1145/1460412.1460472. 30, 87

[61] PHILIP LEVIS AND DAVID CULLER. **Mate: a tiny virtual machine for sensor networks**. *SIGPLAN Not.*, **37**:85–95, October 2002. Available from: http://doi.acm.org/10.1145/605432.605407. 30

[62] R. MULLER, G. ALONSO, AND D. KOSSMANN. **A virtual machine for sensor networks**. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 145–158. ACM, 2007. 30

[63] M. WELSH AND G. MAINLAND. **Programming sensor networks using abstract regions**. NSDI, 2004. 30

[64] PAOLO COSTA, LUCA MOTTOLA, AMY L. MURPHY, AND GIAN PIETRO PICCO. **TeenyLIME: transiently shared tuple space middleware for wireless sensor networks**. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 43–48, New York, NY, USA, 2006. ACM. Available from: http://doi.acm.org/10.1145/1176866.1176874. 30, 45, 148

[65] 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 (TODS)*, **30**(1):122–173, 2005. 30

[66] L.S. BAI, R.P. DICK, AND P.A. DINDA. **Archetype-Based Design: Sensor Network Programming for Application Experts, Not Just Programming Experts**. In *IPSN '09: Proceedings of the 8th international conference on Information processing in sensor networks*, 2009. 30

[67] ATIS ELSTS, JANIS JUDVAITIS, AND LEO SELAVO. **SEAL: A Domain-Specific Language for Novice Wireless Sensor Network Programmers**. In *Software Engineering and Advanced Applications (SEAA), 2013 39th EUROMICRO Conference on*, pages 220–227. IEEE, 2013. 31

[68] **ACM Digital Library**. Available from: http://portal.acm.org/ [cited 08.08.2011.]. 32

[69] **IEEE Xplore Digital Library**. Available from: http://ieeexplore.ieee.org/Xplore/dynhome.jsp [cited 08.08.2011.]. 32

[70] **Elsevier ScienceDirect Scientific Database**. Available from: http://www.sciencedirect.com/ [cited 08.08.2011.]. 32

[71] **SpringerLink integrated full-text database**. Available from: http://www.springerlink.com [cited 08.08.2011.]. 32

[72] ALAN MAINWARING, DAVID CULLER, JOSEPH POLASTRE, ROBERT SZEWCZYK, AND JOHN ANDERSON. **Wireless sensor networks for habitat monitoring**. In *Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, WSNA '02, pages 88–97, New York, NY, USA, 2002. ACM. Available from: http://doi.acm.org/10.1145/570738.570751. 33

[73] WILLIAM MERRILL, FREDRIC NEWBERG, KATHY SOHRABI, WILLIAM KAISER, AND GREG POTTIE. **Collaborative Networking Requirements for Unattended Ground Sensor Systems**. In *Proc. of IEEE Aerospace Conference*, 2003. 33

[74] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Jonathan Hui, and Bruce Krogh. **Energy-efficient surveillance system using wireless sensor networks**. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 270–283, New York, NY, USA, 2004. ACM. Available from: `http://doi.acm.org/10.1145/990064.990096`. 33

[75] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. **Sensor network-based countersniper system**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 1–12, New York, NY, USA, 2004. ACM. Available from: `http://doi.acm.org/10.1145/1031495.1031497`. 33

[76] Bjørn Thorstensen, Tore Syversen, Trond-Are Bjørnvold, and Tron Walseth. **Electronic shepherd - a low-cost, low-bandwidth, wireless network system**. In *Proceedings of the 2nd international conference on Mobile systems, applications, and services*, MobiSys '04, pages 245–255, New York, NY, USA, 2004. ACM. Available from: `http://doi.acm.org/10.1145/990064.990094`. 33

[77] Z. Butler, P. Corke, R. Peterson, and D. Rus. **Virtual fences for controlling cows**. In *Robotics and Automation, 2004. Proceedings. ICRA '04. 2004 IEEE International Conference on*, **5**, pages 4429–4436, april-1 may 2004. 33

[78] Lakshman Krishnamurthy, Robert Adler, Phil Buonadonna, Jasmeet Chhabra, Mick Flanigan, Nandakishore Kushalnagar, Lama Nachman, and Mark 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*, SenSys '05, pages 64–75, New York, NY, USA, 2005. ACM. Available from: `http://doi.acm.org/10.1145/1098918.1098926`. 33, 42

[79] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. **Design and implementation of a sensor network system for vehicle tracking and autonomous interception**. In *Wireless Sensor Networks, 2005. Proceeedings of the Second European Workshop on*, pages 93 – 107, jan.-2 feb. 2005. 33

[80] Sarah Mount, Elena Gaura, Robert M. Newman, Alastair R. Beresford, Sam R. Dolan, and Michael Allen. **Trove: a physical game running on an ad-hoc wireless sensor network**. In *Proceedings of the 2005 joint conference on Smart objects and ambient intelligence: innovative context-aware services: usages and technologies*, sOc-EUSAI '05, pages 235–239, New York, NY, USA, 2005. ACM. Available from: `http://doi.acm.org/10.1145/1107548.1107607`. 34

[81] Loc Ho, Melody Moh, Zachary Walker, Takeo Hamada, and Ching-Fong Su. **A prototype on RFID and sensor networks for elder healthcare: progress report**. In *Proceedings of the 2005 ACM SIGCOMM workshop on Experimental approaches to wireless network design and analysis*, E-WIND '05, pages 70–75, New York, NY, USA, 2005. ACM. Available from: `http://doi.acm.org/10.1145/1080148.1080164`. 34

[82] 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*, pages 1–8. IEEE, 2006. 34

[83] Carl Hartung, Richard Han, Carl Seielstad, and Saxon Holbrook. **FireWxNet: a multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments**. In *Proceedings of the 4th international conference on Mobile systems, applications and services*, MobiSys '06, pages 28–41, New York, NY, USA, 2006. ACM. Available from: `http://doi.acm.org/10.1145/1134680.1134685`. 34

[84] A. Wood, G. Virone, T. Doan, Q. Cao, L. Selavo, Y. Wu, L. Fang, Z. He, S. Lin, and J. Stankovic. **ALARM-NET: Wireless sensor networks for assisted-living and residential monitoring**. Technical report, University of Virginia Computer Science Department, 2006. 34

[85] Liang Liu and Huadong Ma. **Wireless sensor network based mobile pet game**. In *Proceedings of 5th ACM SIGCOMM workshop on Network and system support for games*, NetGames '06, New York, NY, USA, 2006. ACM. Available from: `http://doi.acm.org/10.1145/1230040.1230099`. 34

[86] Joshua Lifton, Mark Feldmeier, Yasuhiro Ono, Cameron Lewis, and Joseph A. Paradiso. **A platform for ubiquitous sensor deployment in occupational and domestic environments**. In *Proceedings of the 6th international conference on Information processing in sensor networks*, IPSN '07, pages 119–127, New York, NY, USA, 2007. ACM. Available from: `http://doi.acm.org/10.1145/1236360.1236377`. 34

[87] V. Santos, P. Bartolomeu, J. Fonseca, and A. Mota. **B-Live - A Home Automation System for Disabled and Elderly People**. In *Industrial Embedded Systems, 2007. SIES '07. International Symposium on*, pages 333 –336, july 2007. 34

[88] Tia Gao, T. Massey, L. Selavo, D. Crawford, Bor rong Chen, K. Lorincz, V. Shnayder, L. Hauenstein, F. Dabiri, J. Jeng, A. Chanmugam, D. White, M. Sarrafzadeh, and M. Welsh. **The Advanced Health and Disaster Aid Network: A Light-Weight Wireless Medical System for Triage**. *Biomedical Circuits and Systems, IEEE Transactions on*, **1**(3):203–216, sept. 2007. 35

[89] J. Wilson, V. Bhargava, A. Redfern, and P. Wright. **A Wireless Sensor Network and Incident Command Interface for Urban Firefighting**. In *Mobile and Ubiquitous Systems: Networking Services, 2007. MobiQuitous 2007. Fourth Annual International Conference on*, pages 1 –7, aug. 2007. 35

[90] B.P. Jarochowski, SeungJung Shin, DaeHyun Ryu, and HyungJun Kim. **Ubiquitous Rehabilitation Center: An Implementation of a Wireless Sensor Network Based Rehabilitation Management System**. In *Convergence Information Technology, 2007. International Conference on*, pages 2349 –2358, nov. 2007. 35

[91] Mateusz Malinowski, Matthew Moskwa, Mark Feldmeier, Mathew Laibowitz, and Joseph A. Paradiso. **CargoNet: a low-cost micropower sensor node exploiting quasi-passive wakeup for adaptive asychronous monitoring of exceptional events**. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 145–159, New York, NY, USA, 2007. ACM. Available from: `http://doi.acm.org/10.1145/1322263.1322278`. 35

[92] Shane B. Eisenman, Emiliano Miluzzo, Nicholas D. Lane, Ronald A. Peterson, Gahng-Seop Ahn, and Andrew T. Campbell. **BikeNet: A mobile sensing system for cyclist experience mapping**. *ACM Trans. Sen. Netw.*, **6**:6:1–6:39, January 2010. Available from: `http://doi.acm.org/10.1145/1653760.1653766`. 35

[93] K. Chebrolu, B. Raman, N. Mishra, P.K. Valiveti, and R. Kumar. **Brimon: a sensor network system for railway bridge monitoring**. In *Proceedings of the 6th international conference on Mobile systems, applications, and services (MobiSys)*, pages 2–14, June 2008. 35

[94] Niclas Finne, Joakim Eriksson, Adam Dunkels, and Thiemo Voigt. **Experiences from two sensor network deployments: self-monitoring and self-configuration keys to success**. In *Proceedings of the 6th international conference on Wired/wireless internet communications*, WWIC'08, pages 189–200, Berlin, Heidelberg, 2008. Springer-Verlag. Available from: `http://portal.acm.org/citation.cfm?id=1788674.1788696`. 35

[95] Changsu Suh, Young-Bae Ko, Cheul-Hee Lee, and Hyung-Joon Kim. **The Design and Implementation of Smart Sensor-based Home Networks**. In *Proc. of the International Symposium on Ubiquitous Computing Systems (UCS'06)*, page 10, 2006. 35

[96] Hui Song, Sencun Zhu, and Guohong Cao. **SVATS: A Sensor-Network-Based Vehicle Anti-Theft System**. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*, pages 2128 –2136, april 2008. 36

[97] Guillermo Barrenetxea, François Ingelrest, Gunnar Schaefer, and Martin Vetterli. **The hitchhiker's guide to successful wireless sensor network deployments**. In *Proceedings of the 6th ACM conference on Embedded network sensor systems*, SenSys '08, pages 43–56, New York, NY, USA, 2008. ACM. Available from: `http://doi.acm.org/10.1145/1460412.1460418`. 36

[98] Nuri Firat Ince, Cheol-Hong Min, Ahmed Tewfik, and David Vanderpool. **Detection of early morning daily activities with static home and wearable wireless sensors**. *EURASIP J. Adv. Signal Process*, **2008**, January 2008. Available from: `http://dx.doi.org/10.1155/2008/273130`. 36

[99] Matteo Ceriotti, Luca Mottola, Gian Pietro Picco, Amy L. Murphy, Stefan Guna, Michele Corra, Matteo Pozzi, Daniele Zonta, and Paolo Zanon. **Monitoring heritage buildings with wireless sensor networks: The Torre Aquila deployment**. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 277–288, Washington, DC, USA, 2009. IEEE Computer Society. Available from: `http://portal.acm.org/citation.cfm?id=1602165.1602191`. 36, 42

[100] Xiaofan Jiang, Stephen Dawson-Haggerty, Prabal Dutta, and David Culler. **Design and implementation of a high-fidelity AC metering network**. In *Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, IPSN '09, pages 253–264, Washington, DC, USA, 2009. IEEE Computer Society. Available from: `http://portal.acm.org/citation.cfm?id=1602165.1602189`. 36, 42

[101] Mo Li and Yunhao Liu. **Underground coal mine monitoring with wireless sensor networks**. *ACM Trans. Sen. Netw.*, **5**:10:1–10:29, April 2009. Available from: `http://doi.acm.org/10.1145/1498915.1498916`. 36

[102] M. Franceschinis, L. Gioanola, M. Messere, R. Tomasi, M.A. Spirito, and P. Civera. **Wireless Sensor Networks for Intelligent Transportation Systems**. In *Vehicular Technology Conference, 2009. VTC Spring 2009. IEEE 69th*, pages 1 –5, april 2009. 36

[103] Carrick Detweiler, Marek Doniec, Mingshun Jiang, Mac Schwager, Robert Chen, and Daniela Rus. **Adaptive decentralized control of underwater sensor networks for modeling underwater phenomena**. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 253–266, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1869983.1870008`. 36

[104] Tsung-te (Ted) Lai, Yu-han (Tiffany) Chen, Polly Huang, and Hao-hua Chu. **PipeProbe: a mobile sensor droplet for mapping hidden pipeline**. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 113–126, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1869983.1869996`. 36

[105] Vladimir Dyo, Stephen A. Ellwood, David W. Macdonald, Andrew Markham, Cecilia Mascolo, Bence Pásztor, Salvatore Scellato, Niki Trigoni, Ricklef Wohlers, and Kharsim Yousef. **Evolution and sustainability of a wildlife monitoring sensor network**. In *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems*, SenSys '10, pages 127–140, New York, NY, USA, 2010. ACM. Available from: `http://doi.acm.org/10.1145/1869983.1869997`. 36

[106] Renjie Huang, Wen-Zhan Song, Mingsen Xu, Nina Peterson, Behrooz Shirazi, and Richard LaHusen. **Real-World Sensor Network for Long-Term Volcano Monitoring: Design and Findings**. *IEEE Transactions on Parallel and Distributed Systems*, **23**(2):321–329, 2012. 36, 98

[107] M. Ceriotti, M. Corrà, L. D'Orazio, R. Doriguzzi, D. Facchin, S. Guna, G.P. Jesi, R.L. Cigno, L. Mottola, A.L. Murphy, et al. **Is There Light at the Ends of the Tunnel? Wireless Sensor Networks for Adaptive Lighting in Road Tunnels**. In *Proceedings of the 10th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN/SPOTS)*, pages 187–198, 2011. 37

[108] Prabal Dutta. **Sustainable Sensing for a Smarter Planet**. *XRDS*, **17**(4):14–20, Summer 2011. 40

[109] Crossbow Technology. **MICA2 Wireless Measurement System datasheet**. Available from: `https://www.eol.ucar.edu/rtf/facilities/isa/internal/CrossBow/DataSheets/mica2.pdf` [cited 20.07.2011.]. 41

[110] *MicaZ mote datasheet*. Available from: `http://www.openautomation.net/uploadsproductos/micaz_datasheet.pdf`. 41

[111] Joseph Polastre, Robert Szewczyk, and David Culler. **Telos: enabling ultra-low power wireless research**. In *Proceedings of the 4th international symposium on Information processing in sensor networks*, IPSN '05, Piscataway, NJ, USA, 2005. IEEE Press. Available from: `http://portal.acm.org/citation.cfm?id=1147685.1147744`. 41, 57

[112] B. Lo, S. Thiemjarus, R. King, and G. Yang. **Body sensor network–a wireless sensor platform for pervasive healthcare monitoring**. In *Adjunct Proceedings of the Third International Conference on Pervasive Computing*, **191**, pages 77–80, 2005. 41, 57

[113] Sentilla. **TMote Mini datasheet**. Available from: http://automatica.dei.unipd.it/public/Schenato/PSC/2010_2011/gruppo4-Building_termo_identification/BibliografiaCasuale/Tmote_Mini_Datasheet.pdf [cited 26.07.2011.]. 41

[114] Texas Instruments. **Single Chip Ultra Low Power RF Transceiver for 315/433/868/915 MHz SRD Band**. Available from: http://www.ti.com/lit/gpn/cc1000 [cited 2011.07.13]. 41

[115] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. **System architecture directions for networked sensors**. *Acm Sigplan Notices*, **35**(11):93–104, 2000. 42

[116] Zach Shelby and Carsten Bormann. *6LoWPAN: The Wireless Embedded Internet*. Wiley Publishing, 2010. Available from: http://portal.acm.org/citation.cfm?id=1824211. 44, 57

[117] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, 1st edition, March 2009. Available from: http://www.amazon.com/TinyOS-Programming-Philip-Levis/dp/0521896061. 45

[118] S. Saruwatari, M. Suzuki, and H. Morikawa. **A compact hard real-time operating system for wireless sensor nodes**. In *Networked Sensing Systems (INSS), 2009 Sixth International Conference on*, pages 1–8, june 2009. 45

[119] A. Eswaran, A. Rowe, and R. Rajkumar. **Nano-RK: an energy-aware resource-centric RTOS for sensor networks**. In *Real-Time Systems Symposium, 2005. RTSS 2005. 26th IEEE International*, pages 265–274, dec. 2005. 45

[120] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. **A dynamic operating system for sensor nodes**. In *Proceedings of the 3rd international conference on Mobile systems, applications, and services*, MobiSys '05, pages 163–176, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1067170.1067188. 45

[121] Jonathan W. Hui and David Culler. **The dynamic behavior of a data dissemination protocol for network programming at scale**. In *Proceedings of the 2nd international conference on Embedded networked sensor systems*, SenSys '04, pages 81–94, New York, NY, USA, 2004. ACM. Available from: http://doi.acm.org/10.1145/1031495.1031506. 45, 69, 147, 148

[122] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. **Timing-sync protocol for sensor networks**. In *Proceedings of the 1st international conference on Embedded networked sensor systems*, SenSys '03, pages 138–149, New York, NY, USA, 2003. ACM. Available from: http://doi.acm.org/10.1145/958491.958508. 48, 115

[123] Bulent Tavli, Kemal Bicakci, Ruken Zilan, and JoseM. Barcelo-Ordinas. **A survey of visual sensor network platforms**. *Multimedia Tools and Applications*, **60**(3):689–726, 2012. 49

[124] Mohammad Rahimi, Rick Baer, Obimdinachi I. Iroezi, Juan C. Garcia, Jay Warrior, Deborah Estrin, and Mani Srivastava. **Cyclops: in situ image sensing and interpretation in wireless sensor networks**. In *Proceedings of the 3rd international conference on Embedded networked sensor systems*, SenSys '05, pages 192–204, New York, NY, USA, 2005. ACM. Available from: http://doi.acm.org/10.1145/1098918.1098939. 49

[125] John Heidemann, Milica Stojanovic, and Michele Zorzi. **Underwater sensor networks: applications, advances and challenges**. *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, **370**(1958):158–175, 2012. 49

[126] JeongGil Ko, Chenyang Lu, Mani B Srivastava, John A Stankovic, Andreas Terzis, and Matt Welsh. **Wireless sensor networks for healthcare**. *Proceedings of the IEEE*, **98**(11):1947–1960, 2010. 49

[127] Luigi Atzori, Antonio Iera, and Giacomo Morabito. **The internet of things: A survey**. *Computer Networks*, **54**(15):2787–2805, 2010. 49

[128] Peter Corke, Tim Wark, Raja Jurdak, Wen Hu, Philip Valencia, and Darren Moore. **Environmental wireless sensor networks**. *Proceedings of the IEEE*, **98**(11):1903–1917, 2010. 49

[129] Mario Gerla and Leonard Kleinrock. **Vehicular networks and the future of the mobile internet**. *Computer Networks*, **55**(2):457–469, 2011. 49

[130] Sarah Clark. **Nokia unveils N9 NFC phone**. Available from: http://www.nfcworld.com/2011/06/21/38138/nokia-unveils-n9-nfc-phone/ [cited 17.09.2011.]. 50

[131] Kit Eathon. **Google's NFC-Powered Digital Wallet: Room For Your Shopping Lists, Credit Cards ... And Complete Trust** . Available from: http://www.fastcompany.com/1755490/google-shopping-wireless-wallet-nfc-payment-nexus-smartphones [cited 17.09.2011.]. 50

[132] IEEE Computer Society. *IEEE Std. 802.11p 2010*. IEEE, amendment 6 edition. Available from: http://standards.ieee.org/getieee802/download/802.11p-2010.pdf. 50

[133] European Space Agency. **Galileo Navigation**. Available from: http://www.esa.int/esaNA/galileo.html [cited 2014-01-28]. 51

[134] SOWNet technologies. **G-Node**. Available from: http://www.sownet.nl/index.php/en/products/gnode [cited 07.02.2012.]. 69

[135] Zolertia. **Z1 Platform**. Available from: http://www.zolertia.com/ti [cited 07.02.2012.]. 69

[136] Gwenhael Goavec-Merou. **SDCard and FAT16 file system implementation for TinyOS http://www.trabucayre.com/page-tinyos.html** [online]. Available from: http://www.trabucayre.com/page-tinyos.html [cited 06.02.2012.]. 69

[137] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. **Protothreads: Simplifying Event-Driven Programming of Memory-Constrained Embedded Systems**. In *Proc. of SenSys'06*, pages 29–42, 2006. 71, 73, 90, 158, 161

[138] Adam Dunkels. **Full TCP/IP for 8-bit architectures**. In *Proceedings of the 1st international conference on Mobile systems, applications and services (MobiSys'03).*, pages 85–98. ACM, 2003. 71

[139] Arduino SA. **Arduino Wireless Shield with XBee Series 2 radios**. Available from: http://arduino.cc/en/Guide/ArduinoWirelessShieldS2 [cited 2014-02-02]. 72

[140] Arduino SA. **Arduino Yún**. Available from: http://arduino.cc/en/Main/ArduinoBoardYun [cited 2014-02-02]. 72

[141] Flutter Wireless. **Wireless ARM development board with over 1 km range**. Available from: http://www.flutterwireless.com/ [cited 2014-02-02]. 72

[142] Robert Faludi. *Building wireless sensor networks: with ZigBee, XBee, Arduino, and Processing*. O'reilly, 2010. 72

[143] Charles E Perkins and Elizabeth M Royer. **Ad-hoc on-demand distance vector routing**. In *Second IEEE Workshop on Mobile Computing Systems and Applications (WMCSA'99).*, pages 90–100. IEEE, 1999. 72

[144] Baptiste Gaultier. **Arduino uIPv6 Stack**. Available from: https://github.com/telecombretagne/Arduino-IPv6Stack/wiki [cited 2014-02-02]. 72

[145] Bill Greiman. **sdfatlib: A FAT16/FAT32 Arduino library for SD/SDHC cards**. Available from: https://code.google.com/p/sdfatlib/ [cited 2014-02-02]. 73

[146] Arduino SA. **Arduino Time Library**. Available from: http://playground.arduino.cc/Code/time [cited 2014-02-02]. 73

[147] Arduino SA. **Interfacing with Hardware: Input**. Available from: http://playground.arduino.cc//Main/InterfacingWithHardware [cited 2014-02-02]. 73

[148] CodeBender team. **codebender**. Available from: https://codebender.cc/ [cited 2014-02-02]. 73

[149] Rolls Royce. **Integrated bridge systems**, 2013. Available from: http://www.rolls-royce.com/marine/products/automation_control/integrated_bridge_systems/ [cited 2014-02-02]. 74

[150] Uni-Safe Electronics. **BNWAS BW-800 Bridge Navigational Watch Alarm System**, 2014. Available from: http://www.unielec.dk/BNWAS_Bridge_Navigational_Watch_Alarm_System-10.htm [cited 2014-02-02]. 74

[151] Maritime and Coastguard Agency. **International Convention for the Safety of Life at Sea**. In *Safety of Navigation*, chapter V. 2002. 74

[152] Furuno. **Installation Manual BRIDGE ALARM SYSTEM BR-1000**, 2009. 74

[153] Ulstein. **Bridge Alarm System**, 2013. Available from: http://www.ulsteingroup.com/kunder/ulstein/cms66.nsf/pages/elcontrl.htm?open&qnfl=flash#electricalandcontrolsystems/electronicsandautomation/ubas/product/ulsteincom.itm [cited 2014-02-02]. 74

[154] Det Norske Veritas. **Classification of Ships Nautical Safety - Offshore Service Vessels**. chapter 20, pages 1–40. 2010. 74

[155] Yushan Pan, Sathiya Kumar Renganayagalu, and Sashidharan Komandur. **Tactile cues for ship bridge operations**. In *27th European Conference on Modelling and Simulation (ECMS'13)*, pages 177–183, Aalesund, 2013. 74

[156] James L. Merlo. **The effects of physiological stress on tactile communication**. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, **50**(16):1562–1566, 2006. 77

[157] Bluetooth special interest group. **Bluetooth specification, Part F: RFCOMM with TS 07.10, Version 1.1**, June 2003. Available from: https://www.bluetooth.org/docman/handlers/DownloadDoc.ashx?doc_id=40909. 79

[158] Bluetooth Special Interest Group. **Specification of the Bluetooth system, Version 4.1**, 2013. Available from: http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Specification+of+the+Bluetooth+system#3. 79

[159] Moslem Amiri. *Measurements of energy consumption and execution time of different operations on Tmote Sky sensor motes*. Master's thesis, Masaryk University, 2010. 82

[160] TIOBE Software. **TIOBE Programming Community Index**, 2014. Available from: http://www.tiobe.com/index.php/content/paperinfo/tpci/index.html [cited 2014-02-10]. 86

[161] Jakob Nielsen. *Usability Engineering*. Morgan Kaufmann, 1st edition, 1993. 86

[162] V.F. Russo. *An object-oriented operating system*. PhD thesis, University of Illinois at Urbana-Chaimpaign, 1991. 87, 165

[163] V.F. Russo, P.W. Madany, and R.H. Campbell. **C++ and operating systems performance: a case study**. Technical report, Department of Computer Science, University of Illinois at Urbana-Champaign, 1991. 87, 166

[164] R. Leavengood. **The Dawn of Haiku - How a volunteer crew brought a crack OS back**. *Spectrum, IEEE*, **49**(5):40–54, 2012. 87

[165] Hauku Inc. **Haiku Operating System**. Available from: http://haiku-os.org/ [cited 2012-06-28]. 87

[166] Doug Simon, Cristina Cifuentes, Dave Cleal, John Daniels, and Derek White. **Java on the bare metal of wireless sensor devices: the squawk Java virtual machine**. In *Proceedings of the 2nd international conference on Virtual execution environments*, VEE '06, pages 78–88, New York, NY, USA, 2006. ACM. Available from: http://doi.acm.org/10.1145/1134760.1134773. 87

[167] Rahul C. Shah, Sumit Roy, Sushant Jain, and Waylon Brunette. **Data MULEs: modeling and analysis of a three-tier architecture for sparse sensor networks**. *Ad Hoc Networks*, **1**(2-3):215 – 233, 2003. Sensor Network Protocols and Applications. Available from: http://www.sciencedirect.com/science/article/pii/S1570870503000039. 100

[168] Northrop Grumman Corporation. **Count Lines Of Code (CLOC)**. Available from: http://cloc.sourceforge.net/ [cited 2012-09-02]. 105

[169] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. **Enabling Large-Scale Storage in Sensor Networks with the Coffee File System**. In *Proceedings of the 8th International Conference on Information Processing in Sensor Networks (IPSN)*, pages 349–360, 2009. 115

[170] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. **A remote code update mechanism for wireless sensor networks**. Technical report, University of California, LA, 2003. 115

[171] Jennifer Yick, Biswanath Mukherjee, and Dipak Ghosal. **Wireless sensor network survey**. *Comput. Netw.*, **52**:2292–2330, August 2008. Available from: http://portal.acm.org/citation.cfm?id=1389582.1389832. 132

[172] Hande Alemdar and Cem Ersoy. **Wireless sensor networks for healthcare: A survey**. *Comput. Netw.*, **54**:2688–2710, October 2010. Available from: http://dx.doi.org/10.1016/j.comnet.2010.05.003. 132

[173] Ning Xu. **A Survey of Sensor Network Applications**. Technical report, University of Southern California, 2002. 132

[174] I. Khemapech, I. Duncan, and A. Miller. **A survey of wireless sensor networks technology**. In *Proc. of The 6th Annual PostGraduate Symposium on The Convergence of Telecommunications, Networking and Broadcasting*, **32**, 2005. 132

[175] I. F. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. **Wireless sensor networks: a survey**. *Computer Networks*, **38**(4):393 – 422, 2002. Available from: http://www.sciencedirect.com/science/article/pii/S1389128601003024. 132

[176] Kumalasari Wardhana and Fabian C Hadipriono. **Analysis of recent bridge failures in the United States**. *Journal of Performance of Constructed Facilities*, **17**(3):144–150, 2003. 134

[177] K. Terfloth, G. Wittenburg, and J. Schiller. **FACTS: a rule-based middleware architecture for wireless sensor networks**. In *Proc. of the 1st Int. Conf. on Communication System Software and Middleware (COMSWARE)*, 2006. 148

[178] Joan Daemen and Vincent Rijmen. *The design of Rijndael: AES-the advanced encryption standard.* Springer, 2002. 162

Appendices

A WSN deployments

A.1 Application taxonomy

Wireless sensors networks contain huge potential for wide range of life quality improvement applications. To formalize the application and deployment design space, a taxonomy is established. It is an adapted version of multiple previous research papers [171, 172, 173, 174, 175]. In the presented taxonomy, deployments are divided not by their technical characteristics (high/low sampling rate, single/multi hop network, etc), rather by the application field.

The author suggests the following taxonomy:

1 Environmental monitoring:

  1.1 Habitat and weather monitoring,

  1.2 Environmental forecasting:

    1.2.1 Forest fire detection,

    1.2.2 Flood and glacier detection,

    1.2.3 Volcano monitoring,

  1.3 Precision agriculture,

  1.4 Underwater networks,

2 Animal monitoring:

  2.1 Wild animal monitoring,

  2.2 Domestic animal monitoring and control,

3 Human-centric applications:

  3.1 Human health telemonitoring,

  3.2 Human monitoring in emergency situations,

    3.3 Human indoor tracking,
    3.4 Firefighter and police assistance systems,
    3.5 Medication intake accounting,
    3.6 Daily activity recognition,
    3.7 Child education and sensor games,

4 Infrastructure monitoring:

    4.1 Heritage buildings and sites,
    4.2 Civil infrastructure monitoring:
        4.2.1 Bridge monitoring,
        4.2.2 Tunnel monitoring,
    4.3 Power line and water pipe monitoring,
    4.4 Security systems,

5 Asset tracking:

    5.1 Anti-theft systems,
    5.2 Good and daily object tracking,

6 Smart buildings:

    6.1 Home/office automation,
    6.2 Smart energy usage,

7 Military applications:

    7.2 Battlefield surveillance,
    7.3 Opposing force investigation,

8 Urban applications:

    8.1 Vehicle tracking and traffic monitoring,
    8.2 City environment monitoring,

9 Industrial applications:

    9.1 Industrial equipment monitoring and control,
    9.2 Coal mine monitoring,

10 Smart user interfaces and art.

Environmental monitoring and is the most popular application class of sensor networks, therefore it is divided more specifically. Human centric sensor networks is also a large class, containing mostly medical applications. Infrastructure monitoring has became popular due to several significant problems faced in real life, including bridge failures [176]. Military research was the initial driver for sensor network evolution, however, the published applications represent only friendly force, battlefield and enemy monitoring.

A.2 Deployment survey detailed results

Table A.1: Deployments: deployment state and attributes

Nr	Codename	Deployment state	Mote count	Heterog. motes	Base station count	Base station hardware
1	Habitats	pilot	32	n	1	Mote + PC with satellite link to Internet
2	Minefield	pilot	20	n	0	All motes capable to connect to a PC via Ethernet
3	Battlefield	prototype	70	y (soft, by role)	1	Mote + PC
4	Line in the sand	pilot	90	n	1	Root connects to long range radio relay
5	Counter-sniper	prototype	56	n	1	Mote + PC
6	Electro-shepherd	pilot	180	y	1+	Mobile mote
7	Virtual fences	prototype	8	n	1	Laptop
8	Oil tanker	pilot	26	n	4	Stargate Gateway + Intel Mote, wall powered.
9	Enemy vehicles	pilot	100	y	1	Mobile powermotes - laptop on wheels
10	Trove game	pilot	10	n	1	Mote + PC
11	Elder RFID	prototype	3	n	1	Mote + PC
12	Murphy potatoes	pilot	109	n	1	Stargate Gateway + Tnode, solar panel,
13	Firewxnet	pilot	13	n	1 BS + 5 gateways	Gateway: Soekris net4801 with Gentoo Linux and Trango Access5830 long-range 10Mbps wireless; BS: PC with satellite link 512/128Kbps
14	AlarmNet	prototype	15	y	varies	Stargate Gateway with MicaZ, wall powered.
15	Ecuador Volcano	pilot	19	y	1	Mote + PC
16	Pet game	prototype	?	n	1+	Mote + MIB510 board + PC
17	Plug	pilot	35	n	1	Mote + PC
18	B-Live	pilot	10+	y	1	B-Live modules connected to PC, wheelchair computer etc
19	Biomotion	pilot	25	n	1	Mote + PC
20	AID-N	pilot	10	y	1+	Mote + PC
21	Firefighting	prototype	20	y	1+	?

...

Table A.1 – continued

Nr	Codename	Deployment state	Mote count	Heterog. motes	Base station count	Base station hardware
22	Rehabil	prototype	?	y	1	Mote + PC
23	CargoNet	pilot	<10	n	1+	Mote + PC?
24	Fence monitor	prototype	10	n	1	Mote + PC?
25	BikeNet	prototype	5	n	7+	802.15.4/Bluetooth bridge + Nokia N80 OR mote + Aruba AP-70 embedded PC
26	BriMon	prototype	12	n	1	Mobile Train TMote, static Bridge Tmotes
27	IP net	pilot	25	n	1	Mote + PC?
28	Smart home	prototype	12	y	1	EMPOSII embedded PC with touchscreen, internet, wall powered
29	SVATS	prototype	6	n	1	?
30	Hitchhiker	pilot?	16		1	?
31	Daily morning	prototype	1	n	1	Mote + MIB510 board + PC
32	Heritage	stable	17	y	1	3Mate mote + Gumstix embedded PC with SD card and WiFi
33	AC meter	pilot	49	n	2+	Meraki Mini and the OpenMesh Mini-Router wired together with radio
34	Coal mine	prototype	27	n	1	?
35	ITS	prototype	8	n	1	?
36	Underwater	prototype	4	n	0	-
37	PipeProbe	prototype	1	n	1	Mote + PC
38	Badgers	stable	74 mobile + 26? static	y	1+	Mote
39	Helens volcano	pilot	13	n	1	?
40	Tunnels	pilot	40	n	2	Mote + Gumstix Verdex Pro

Table A.2: Deployments: sensing

Nr	Codename	Sensors	Periodic or event-based function	Sampling rate, Hz	GPS used
1	Habitats	temperature, light, barometric pressure, humidity, and passive infrared	periodic	0.0166667	n
2	Minefield	sound, magnetometer, accelerometers, voltage, imaging	periodic	?	y
3	Battlefield	magnetometer	event	10	n
4	Line in the sand	magnetometer, radar	event	?	n
5	Counter-sniper	sound	event	1000000	n
6	Electro-shepherd	temperature	periodic	?	y
7	Virtual fences	-	both	?	y
8	Oil tanker	accelerometer	periodic	19200	n

...

Table A.2 – continued

Nr	Codename	Sensors	Periodic or event-based function	Sampling rate, Hz	GPS used
9	Enemy vehicles	magnetometer, ultrasound transceiver	event	?	y, on power nodes
10	Trove game	accelerometers, light	periodic	?	n
11	Elder RFID	RFID reader	periodic	1	n
12	Murphy potatoes	temperature, humidity	periodic	0.0166667	n
13	Firewxnet	temperature, humidity, wind speed and direction	periodic	0.8333333	n
14	AlarmNet	motion, blood pressure, body scale, dust, temperature, light	both	≤ 1Hz	n
15	Ecuador Volcano	seismometers, acoustic	both	100	y, on BS
16	Pet game	temperature, light, sound	periodic	configurable	n
17	Plug	sound, light, electric current, voltage, vibration, motion, temperature	periodic	8000	n
18	B-Live	light, electric current, switches	event	?	n
19	Biomotion	accelerometer, gyroscope, capacitive distance sensor	periodic	100	n
20	AID-N	pulse oximeter, ECG, blood pressure, heart beat	periodic	depends on queries	n
21	Firefighting	temperature	periodic	?	n
22	Rehabil	temperature, humidity, light	periodic	?	n
23	CargoNet	shock, light, magnetic switch, sound, tilt, temperature, humidity	both	0.0166667	n
24	Fence monitor	accelerometer	event	10	n
25	BikeNet	magnetometer, pedal speed, inclinometer, lateral tilt, GSR stress, speedometer, CO2, sound, GPS	both	configurable	y
26	BriMon	accelerometer	periodic	0.6666667	n
27	IP net	temperature, luminosity, vibration, microphone, movement detector	event	?	n
28	Smart home	Ligth, temperature, humidity, air pressure, acceleration, gas leak, motion	both	?	n
29	SVATS	radio RSSI	periodic	?	n
30	Hitchhiker	air temperature and humidity, surface temperature, solar radiation, wind speed and direction, soil water content and suction, and precipitation	periodic	?	n
31	Daily morning	accelerometer	periodic	50	n
32	Heritage	fiber optic deformation, accelerometers, analog temperature	periodic	200	n
33	AC meter	current	both	≤ 14000	n
34	Coal mine	- (sense radio neighbors only)	periodic	-	n
35	ITS	anisotropic magneto-resistive and pyroelectric	event	varies	n
36	Underwater	pressure, temperature, CDOM, salinity, dissolved oxygen, cameras; motor actuator	periodic	≤1Hz	n
37	PipeProbe	gyroscope, pressure	periodic	33	n
38	Badgers	humidity, temperature	event	?	n
39	Helens volcano	geophone, accelerometer	periodic	100000?	y

...

Table A.2 – continued

Nr	Codename	Sensors	Periodic or event-based function	Sampling rate, Hz	GPS used
40	Tunnels	light, temperature, voltage	periodic	0.0333333	n

Table A.3: Deployments: lifetime and energy

Nr	Codename	Lifetime, days	Energy source	Sleep time, sec	Duty cycle, %	Power-motes present?
1	Habitats	270	battery	60	?	yes, gateways
2	Minefield	?	battery	?	?	yes, all
3	Battlefield	5-50	battery	varies	varies	yes, base station
4	Line in the sand	?	battery and solar	?	?	yes, root
5	Counter-sniper	?	battery	0	100	no
6	Electro-shepherd	50	battery	?	< 1	no
7	Virtual fences	2h 40min	battery	0	100	no
8	Oil tanker	82	battery	64800	< 1	yes, gateways
9	Enemy vehicles	?	battery	?	?	yes, mobile nodes
10	Trove game	?	battery	?	?	yes, base station
11	Elder RFID	?	battery	0?	100?	yes, base station
12	Murphy potatoes	21	battery	60	11	yes, base station
13	Firewxnet	21	battery	840	6.67	yes, gateways
14	AlarmNet	?	battery	?	configuration dependent	yes, base stations
15	Ecuador Volcano	19	battery	0	100	yes, base station
16	Pet game	?	battery	?	?	yes, base station
17	Plug	-	power-net	0	100	yes, all
18	B-Live	-	battery	0	100	yes, all
19	Biomotion	5 hours	battery	0	100	yes, base stations
20	AID-N	6	battery	0	100	yes, base station
21	Firefighting	4+	battery	0	100	yes, infrastructure motes
22	Rehabil	?	battery	?	?	yes, base station
23	CargoNet	1825	battery	varies	0.001	no
24	Fence monitor	?	battery	1	?	yes, base station
25	BikeNet	?	battery	?	?	yes, gateways
26	BriMon	625	battery		0.55	no
27	IP net	?	battery	?	20	yes, base station
28	Smart home	?	battery	?	?	yes
29	SVATS	unlimited	power-net	not implemented	-	yes, all
30	Hitchhiker	60	battery and solar	5	10	yes, base station
31	Daily morning	?	battery	0?	100?	yes, base station
32	Heritage	525	battery	0.57	0.05	yes, base station
33	AC meter	?	power-net	?	?	yes, gateways
34	Coal mine	?	battery	?	?	yes, base station?
35	ITS	?	power-net?	0?	100?	yes, all

...

Table A.3 – continued

Nr	Codename	Lifetime, days	Energy source	Sleep time, sec	Duty cycle, %	Power-motes present?
36	Underwater	?	battery	?	?	no
37	PipeProbe	4 hours	battery	0	100	yes, base station
38	Badgers	7	battery	?	0.05	no
39	Helens volcano	400	battery	0?	100?	yes, all
40	Tunnels	480	battery	0.25	?	yes, base stations

Table A.4: Deployments: used motes and radio chips

Nr	Codename	Mote	Ready or custom	Mote motivation	Radio chip	Radio protocol
1	Habitats	Mica	adapted	custom Mica weather board and packaging	RFM TR1000	?
2	Minefield	WINS NG 2.0	custom	need for high performance	?	?
3	Battlefield	Mica2	adapted	energy and bandwidth efficient, simple and flexible	Chipcon CC1000	SmartRF
4	Line in the sand	Mica2	adapted	?	Chipcon CC1000	SmartRF
5	Counter-sniper	Mica2	adapted	?	Chipcon CC1000	SmartRF
6	Electro-shepherd	Custom + Active RFID tags	custom	packaging adapted to sheep habits	unnamed UHF	?
7	Virtual fences	Zaurus PDA	ready	off-the-shelf	unnamed Wifi	802.11
8	Oil tanker	Intel Mote	adapted	?	Zeevo TC2001P	Bluetooth 1.1
9	Enemy vehicles	Mica2Dot	adapted	?	Chipcon CC1000	SmartRF
10	Trove game	Mica2	ready	off-the-shelf	Chipcon CC1000	SmartRF
11	Elder RFID	Mica2	adapted	off-the-shelf, RFID reader added	Chipcon CC1000 + RFID	SmartRF + RFID
12	Murphy potatoes	TNOde, Mica2 like	custom	packaging + sensing	Chipcon CC1000	SmartRF
13	Firewxnet	Mica2	adapted	MANTIS support, AA batteries, easily extensible	Chipcon CC1000	SmartRF
14	AlarmNet	Mica2 + TMote Sky	adapted	off-the-shelf, extensible	Chipcon CC1000	SmartRF
15	Ecuador Volcano	Tmote Sky	adapted	off-the-shelf	Chipcon CC2420	802.15.4
16	Pet game	MicaZ	ready	off-the-shelf	Chipcon CC2420	802.15.4
17	Plug	Plug Mote	custom	specific sensing + packaging	Chipcon CC2500	?

...

Table A.4 – continued

Nr	Codename	Mote	Ready or custom	Mote motivation	Radio chip	Radio protocol
18	B-Live	B-Live module	custom	custom modular system	?	?
19	Biomotion	custom	custom	size constraints	Nordic nRF2401A	-
20	AID-N	TMote Sky + MicaZ	adapted	off the shelf, extensible	Chipcon CC2420	802.15.4
21	Firefighting	TMote Sky	adapted	off the shelf, easy prototyping	Chipcon CC2420	802.15.4
22	Rehabil	Maxfor TIP 7xxCM: TelosB-compatible	ready	off-the-shelf	Chipcon CC2420	802.15.4
23	CargoNet	CargoNet mote	custom	low power, low cost components	Chipcon CC2500	-
24	Fence monitor	Scatterweb ESB	ready	off-the-shelf	Chipcon CC1020	?
25	BikeNet	TMote Invent	adapted	off-the-shelf mote providing required connectivity	Chipcon CC2420	802.15.4
26	BriMon	Tmote Sky	adapted	off the shelf	Chipcon CC2420	802.15.4
27	IP net	Scatterweb ESB	adapted	Necessary sensors onboard	TR1001	?
28	Smart home	ZigbeX	custom	specific sensor, size and power constraints	Chipcon CC2420	802.15.4
29	SVATS	Mica2	ready	off-the-shelf	Chipcon CC1000	SmartRF
30	Hitchhiker	TinyNode	adapted	long range communication	Semtech XE1205	?
31	Daily morning	MicaZ	ready	off-the-shelf	Chipcon CC2420	802.15.4
32	Heritage	3Mate!	adapted	TinyOS supported mote with custom sensors	Chipcon CC2420	802.15.4
33	AC meter	ACme (Epic core)	adapted	modular, convenient prototyping	Chipcon CC2420	802.15.4
34	Coal mine	Mica2	ready	off-the-shelf	Chipcon CC1000	SmartRF
35	ITS	Custom	custom	specific sensing needs	Chipcon CC2420	802.15.4
36	Underwater	AquaNode	custom	specific packaging, sensor and actuator needs	custom	-
37	PipeProbe	Eco mote	adapted	size and energy constraints	Nordic nRF24E1	?
38	Badgers	V1: Tmote Sky + ext. board; V2: custom	v1: adapted, v2: custom	v1: off-the-shelf, v2: optimizations	Atmel AT86RF230	802.15.4
39	Helens volcano	custom	custom	specific computational, sensing and packaging needs	Chipcon CC2420	802.15.4
40	Tunnels	TRITON mote: TelosB-like	custom	reuse and custom packaging	Chipcon CC2420	802.15.4

Table A.5: Deployments: used microcontrollers

Nr	Codename	MCU count	MCU Name	Arch., bits	MHz	RAM, KB	Prog. Mem., KB
1	Habitats	1	Atmel ATMega103L	8	4	4	128
2	Minefield	1	Hitachi SH4 7751	32	167	64000	0
3	Battlefield	1	Atmel ATMega128	8	7.3	4	128
4	Line in the sand	1	Atmel ATMega128	8	4	4	128
5	Counter-sniper	1 + FPGA	Atmel ATMega128L	8	7.3	4	128
6	Electro-shepherd	1	Atmel ATMega128	8	7.3	4	128
7	Virtual fences	1	Intel StrongArm	32	206	65536	?
8	Oil tanker	1	Zeevo ARM7TDMI	32	12	64	512
9	Enemy vehicles	1	Atmel ATMega128L	8	4	4	128
10	Trove game	1	Atmel ATMega128	8	7.3	4	128
11	Elder RFID	1	Atmel ATMega128	8	7.3	4	128
12	Murphy potatoes	1	Atmel ATMega128L	8	8	4	128
13	Firewxnet	1	Atmel ATMega128L	8	7.3	4	128
14	AlarmNet	1	Atmel ATMega128L	8	7.3	4	128
15	Ecuador Volcano	1	TI MSP430F1611	16	8	10	48
16	Pet game	1	Atmel ATMega128	8	7.3	4	128
17	Plug	1	Atmel AT91SAM7S64	32	48	16	64
18	B-Live	2	Microchip PIC18F2580	8	40	1.5	32
19	Biomotion	1	TI MSP430F149	16	8	2	60
20	AID-N	1	TI MSP430F1611	16	8	10	48
21	Firefighting	1	TI MSP430F1611	16	8	10	48
22	Rehabil	1	TI MSP430F1611	16	8	10	48
23	CargoNet	1	TI MSP430F135	16	8?	0.512	16
24	Fence monitor	1	TI MSP430F1612	16	7.3	5	55
25	BikeNet	1	TI MSP430F1611	16	8	10	48
26	BriMon	1	TI MSP430F1611	16	8	10	48
27	IP net	1	TI MSP430F149	16	8	2	60
28	Smart home	1	Atmel ATMega128	8	8	4	128
29	SVATS	1	Atmel ATMega128L	8	7.3	4	128
30	Hitchhiker	1	TI MSP430F1611	16	8	10	48
31	Daily morning	1	Atmel ATMega128	8	7.3	4	128
32	Heritage	1	TI MSP430F1611	16	8	10	48
33	AC meter	1	TI MSP430F1611	16	8	10	48
34	Coal mine	1	Atmel ATMega128	8	7.3	4	128
35	ITS	2	ARM7 + MSP430F1611	32 + 8	? + 8MHz	64 + 10	? + 48
36	Underwater	1	NXP LPC2148 ARM7TDMI	32	60	40	512
37	PipeProbe	1	Nordic nRF24E1 DW8051	8	16	4.25	32
38	Badgers	1	Atmel ATMega128V	8	8	8	128
39	Helens volcano	1	Intel XScale PXA271	32	13 (624 max)	256	32768
40	Tunnels	1	TI MSP430F1611	16	8	10	48

Table A.6: Deployments: external memory

Nr	Codename	Available ext.mem., KB	SD/MMC used	Ext. mem. used	File system used
1	Habitats	512	n	y	n
2	Minefield	16000	n	y	y
3	Battlefield	512	n	n	n
4	Line in the sand	512	n	n	?
5	Counter-sniper	512	n	n	n
6	Electro-shepherd	512	n	y	n
7	Virtual fences	?	y	y	y
8	Oil tanker	0	n	n	n
9	Enemy vehicles	512	n	n	n
10	Trove game	512	n	n	n
11	Elder RFID	512	n	n	n
12	Murphy potatoes	512	n	n	n
13	Firewxnet	512	n	n	n
14	AlarmNet	512	n	n	n
15	Ecuador Volcano	1024	n	y	n
16	Pet game	512	n	n	n
17	Plug	0	n	n	n
18	B-Live	0	n	n	n
19	Biomotion	0	n	n	n
20	AID-N	1024	n	n	n
21	Firefighting	1024	n	n	n
22	Rehabil	1024	n	n	n
23	CargoNet	1024	n	y	n
24	Fence monitor	0	n	n	n
25	BikeNet	1024	n	y?	n
26	BriMon	1024	n	y	n
27	IP net	1024	n	n	n
28	Smart home	512	n	?	n
29	SVATS	512	n	n	n
30	Hitchhiker	1024	n	n	n
31	Daily morning	512	n	n	n
32	Heritage	1024	n	n	n
33	AC meter	2048	n	y	n
34	Coal mine	512	n	n	n
35	ITS	?	n?	?	n?
36	Underwater	2097152?	y	y	n
37	PipeProbe	0	n	n	n
38	Badgers	2097152	y	y	n
39	Helens volcano	0	n	n	n
40	Tunnels	1024	n	n	n

Table A.7: Deployments: sensor and user interface

Nr	Codename	Sensor interface	User Interface
1	Habitats	soft-I2C	3 LEDs
2	Minefield	ADC + ?	?
3	Battlefield	ADC	3 LEDs

...

Table A.7 – continued

Nr	Codename	Sensor interface	User Interface
4	Line in the sand	ADC	3 LEDs
5	Counter-sniper	ADC + ?	3 LEDs
6	Electro-shepherd	1-wire	?
7	Virtual fences	-	GUI
8	Oil tanker	SPI	1 LED
9	Enemy vehicles	ADC + ?	1 LED
10	Trove game	ADC	3 LEDs, buzzer
11	Elder RFID	UART	3 LEDs
12	Murphy potatoes	soft-I2C	3 LEDs
13	Firewxnet	ADC	3 LEDs
14	AlarmNet	ADC + UART + GPIO	3 LEDs, color LCD
15	Ecuador Volcano	SPI?	3 LEDs
16	Pet game	ADC?	3 LEDs
17	Plug	?	2 LEDs
18	B-Live	?	?
19	Biomotion	ADC	3 LEDs
20	AID-N	UART, ADC	5 LEDs, 2x8 text LCD, 4 User buttons
21	Firefighting	?	3 LEDs
22	Rehabil	ADC, soft-I2C	3 LEDs
23	CargoNet	ADC, GPIO, soft-I2C	-
24	Fence monitor	ADC	1 LED
25	BikeNet	UART, ADC, GPIO	3 LEDs
26	BriMon	ADC	3 LEDs
27	IP net	?	2+ LEDs
28	Smart home	I2C, ADC, GPIO	?
29	SVATS	ADC	3 LEDs
30	Hitchhiker	ADC, soft-I2C, GPIO	1 LED
31	Daily morning	ADC	3 LEDs
32	Heritage	ADC + SPI?	3 LEDs
33	AC meter	SPI	3 LEDs
34	Coal mine	-	3 LEDs
35	ITS	?	?
36	Underwater	?	?
37	PipeProbe	ADC, SPI	-
38	Badgers	I2C	3 LEDs
39	Helens volcano	SPI?	-
40	Tunnels	ADC, I2C	3 LEDs

Table A.8: Deployments: communication

Nr	Codename	Report rate, 1/h	Payload size, B	Radio range, m	Speed, kbps	Connectivity type
1	Habitats	60	?	200 (1200 with Yagi 12dBi)	40	connected
2	Minefield	?	?	?	?	connected
3	Battlefield	?	?	300	38.4	intermittent
4	Line in the sand	?	1	300	38.4	connected
5	Counter-sniper	?	?	60	38.4	connected
6	Electro-shepherd	0.33	7+	150-200m	?	connected

. . .

Table A.8 – continued

Nr	Codename	Report rate, 1/h	Payload size, B	Radio range, m	Speed, kbps	Connectivity type
7	Virtual fences	1800	8?	?	54000	connected
8	Oil tanker	0.049	?	30	750	connected
9	Enemy vehicles	1800	?	30	38.4	connected
10	Trove game	?	?	?	38.4	connected
11	Elder RFID	?	19	?	38.4	connected
12	Murphy potatoes	6	22		76.8	connected
13	Firewxnet	200	?	400	38.4	intermittent
14	AlarmNet	depends on config.	29	?	38.4	connected
15	Ecuador Volcano	depends on events	16	1000	250	connected
16	Pet game	depends on config.	?	100	250	connected
17	Plug	720	21	?	?	connected
18	B-Live	-	?	?	?	connected
19	Biomotion	360000	16	15	1000	connected
20	AID-N	depends on queries	?	66	250	connected
21	Firefighting	?	?	20	250	connected
22	Rehabil	?	12	30	250	connected
23	CargoNet	depends on events	?	?	250	sporadic
24	Fence monitor	?	?	300	76.8	connected
25	BikeNet	opportunistic	?	20	250	sporadic
26	BriMon	62	116	125	250	sporadic
27	IP net	?	?	300	19.2	connected
28	Smart home	?	?	75-100 outdoor/20-30 indoor	250	connected
29	SVATS	?	?	400	38.4	connected
30	Hitchhiker	?	24	500	76.8	connected
31	Daily morning	180000	2?	100	250	connected
32	Heritage	6	?	125	250	intermittent
33	AC meter	60 default (configurable)	?	125	250	connected
34	Coal mine	?	7	4m forced, 20m max	38.4	intermittent
35	ITS	varies	5*n	?	250	connected
36	Underwater	900	11	?	0.3	intermittent
37	PipeProbe	72000	?	10	1000	connected
38	Badgers	2380+	10	1000	250	connected
39	Helens volcano	depends on config.	?	9600	250	connected
40	Tunnels	120	?	?	250	connected

Table A.9: Deployments: communication media

Nr	Codename	Communication media	Used channels	Directinality used?
1	Habitats	radio over air	1	n
2	Minefield	radio over air + sound over air	?	n
3	Battlefield	radio over air	1	n
4	Line in the sand	radio over air	1	n
5	Counter-sniper	radio over air	1	n
6	Electro-shepherd	radio over air	?	n
7	Virtual fences	radio over air	2	y
8	Oil tanker	radio over air	79	n
9	Enemy vehicles	radio over air	1	n
10	Trove game	radio over air	1	n
11	Elder RFID	radio over air	1	n
12	Murphy potatoes	radio over air	1	n
13	Firewxnet	radio over air	1	y, gateways
14	AlarmNet	radio over air	1	n
15	Ecuador Volcano	radio over air	1	y
16	Pet game	radio over air	1	n
17	Plug	radio over air	?	n
18	B-Live	wire mixed with radio over air	?	n
19	Biomotion	radio over air	1	n
20	AID-N	radio over air	1	n
21	Firefighting	radio over air	4	n
22	Rehabil	radio over air	1?	n
23	CargoNet	radio over air	1	n
24	Fence monitor	radio over air	1	n
25	BikeNet	radio over air	1	n
26	BriMon	radio over air	16	n
27	IP net	radio over air	1	n
28	Smart home	radio over air	16	?
29	SVATS	radio over air	?	n
30	Hitchhiker	radio over air	1	n
31	Daily morning	radio over air	1	n
32	Heritage	radio over air	1	n
33	AC meter	radio over air	1	n
34	Coal mine	radio over air	1	n
35	ITS	radio over air	1	n
36	Underwater	ultra-sound over water	1	n
37	PipeProbe	radio over air and water	1	n
38	Badgers	radio over air	?	n
39	Helens volcano	radio over air	1?	y
40	Tunnels	radio over air	2	n

Table A.10: Deployments: network

Nr	Codename	Network topology	Mobile motes?	Deployment area	Max hop count	Randomly deployed?
1	Habitats	multi-one-hop	n	1000 x 1000m	1	n
2	Minefield	mesh	y	30 x 40m	?	y
3	Battlefield	mesh	n	85m long road	?	y
4	Line in the sand	mesh	n	18 x 8m	?	n
5	Counter-sniper	mesh	n	30 x 15m	11	y
6	Electro-shepherd	one-hop	y	?	1	y (attached to animals)
7	Virtual fences	mesh	y	300 x 300m	5	y (attached to animals)
8	Oil tanker	multi-mesh	n	150 x 100m	?	n
9	Enemy vehicles	mesh	y, power node	20 x 20m	6	n
10	Trove game	one-hop	y	?	1	y, attached to users
11	Elder RFID	one-hop	n (mobile RFID tags)	$< 10m^2$	1	n
12	Murphy potatoes	mesh	n	1000 x 1000m	10	n
13	Firewxnet	multi-mesh	n	$160km^2$	4?	n
14	AlarmNet	mesh	y, mobile body motes	apartment	?	n
15	Ecuador Volcano	mesh	n	8000 x 1000m	6	n
16	Pet game	mesh	y	?	?	y
17	Plug	mesh	n	40 x 40	?	n
18	B-Live	multi-one-hop	n	house	2	n
19	Biomotion	one-hop	y, mobile body motes	room	1	n (attached to predefined body parts)
20	AID-N	mesh	y	?	1+	y, attached to users
21	Firefighting	predefined tree	y, human mote	$3200m^2$	?	n
22	Rehabil	one-hop	y, human motes	gymnastics room	1	y, attached to patients and training machines
23	CargoNet	one-hop	y	truck, ship or plane	1	n
24	Fence monitor	one-hop?	n	35 x 2m	1?	n
25	BikeNet	mesh	y	5km long track	?	y (attached to bicycles)
26	BriMon	multi-mesh	y, mobile BS	2000 x 1	4	n
27	IP net	multi-one-hop	n	250x25 3 story building + mock-up town $500m^2$	?	n
28	Smart home	one-hop	n	?	?	n
29	SVATS	mesh	y, motes in cars	parking place	?	n
30	Hitchhiker	mesh	n	500 x 500m	2?	n
31	Daily morning	one-hop	y, body mote	house	1	n (attached to human)
32	Heritage	mesh	n	7.8x4.5x26m	6	n (initial deployment static, but can be moved later)

...

Table A.10 – continued

Nr	Codename	Network topology	Mobile motes?	Deployment area	Max hop count	Randomly deployed?
33	AC meter	mesh	n	building	?	y (Given to users who plug in power outlets of their choice)
34	Coal mine	multi-path mesh	n	8 x 4 x ? m	?	n
35	ITS	mesh	n	140m long road	7?	n
36	Underwater	mesh	y	?	1	n
37	PipeProbe	one-hop	y	0.18 x 1.40 x 3.45m	1	n
38	Badgers	mesh	y	1000 x 2000m ?	?	y (attached to animals)
39	Helens volcano	mesh	n	?	1+?	n
40	Tunnels	multi-mesh	n	230m long tunnel	4	n

Table A.11: Deployments: networking protocol stack

Nr	Codename	Custom MAC	TDMA or CSMA	Routing used	Custom routing	Reactive or proactive routing	IPv6 used	QoS: safe delivery	QoS: data priorities
1	Habitats	n	CSMA	n	-	-	n	n	n
2	Minefield	n	CSMA	y	?	?	?	?	?
3	Battlefield	y	CSMA	y	y	proactive	n	y	?
4	Line in the sand	y	CSMA	y	y	proactive	n	y	n
5	Counter-sniper	n	CSMA	y	y	proactive	n	n	-
6	Electro-shepherd	y	CSMA	-	-	-	n	y	n
7	Virtual fences	n	CSMA	y	n	-	IPv4?	n	n
8	Oil tanker	n	CSMA	y	n	-	n	y	n
9	Enemy vehicles	y	CSMA	y	y	proactive	n	n	-
10	Trove game	n	CSMA	n	-	-	n	n	n
11	Elder RFID	n	CSMA	n	-	-	n	n	n
12	Murphy potatoes	y	CSMA	y	n	proactive	n	n	n
13	Firewxnet	y	CSMA	y	y	proactive	n	y	n
14	AlarmNet	y	CSMA	y	n	?	n	y	y
15	Ecuador Volcano	n	CSMA	y	y	proactive	n	y	n
16	Pet game	n	CSMA	y	n	?	n	n	n
17	Plug	y	CSMA	y	y	?	n	n	n
18	B-Live	?	?	n	-	-	n	?	?
19	Biomotion	y	TDMA	n	-	-	n	n	n
20	AID-N	?	?	y	n	proactive	n	y	n
21	Firefighting	n	CSMA	y, static	n	proactive	n	n	n
22	Rehabil	n	CSMA	n	-	-	n	n	n
23	CargoNet	y	CSMA	n	-	-	n	n	n

...

Table A.11 – continued

Nr	Codename	Custom MAC	TDMA or CSMA	Routing used	Custom routing	Reactive or proactive routing	IPv6 used	QoS: safe delivery	QoS: data priorities
24	Fence monitor	n	CSMA?	y	y	proactive?	n	n	n
25	BikeNet	y	CSMA	y	y	reactive	n	y	n
26	BriMon	y	TDMA	y	y	proactive	n	y	n
27	IP net	n	CSMA	y	y	proactive	?	?	?
28	Smart home	?	?	y	?	?	n	?	?
29	SVATS	n	CSMA	y	n	?	n	n	n
30	Hitchhiker	y	TDMA	y	y	reactive	n	y	n
31	Daily morning	n	CSMA	n	-	-	n	n	n
32	Heritage	y	TDMA	y	y	proactive	n	y	y
33	AC meter	n	?	y	n	proactive	y	y	n
34	Coal mine	n	CSMA	y	y	proactive	n	y	n
35	ITS	y?	CSMA?	y	y	reactive	n	y	n
36	Underwater	y	TDMA	n	-	-	n	n	n
37	PipeProbe	n	?	n	-	-	n	n	n
38	Badgers	n	CSMA	y	y	proactive	y	n	y
39	Helens volcano	y	TDMA	y	?	?	n	y	y
40	Tunnels	n	CSMA	y	y	proactive	n	n	n

Table A.12: Deployments: used operating system and middleware

Nr	Codename	OS used	Self-made OS	Middleware used
1	Habitats	TinyOS	n	
2	Minefield	customized Linux	n	
3	Battlefield	TinyOS	n	
4	Line in the sand	TinyOS	n	
5	Counter-sniper	TinyOS	n	
6	Electro-shepherd	?	y	
7	Virtual fences	Linux	n	
8	Oil tanker	?	n	
9	Enemy vehicles	TinyOS	n	
10	Trove game	TinyOS	n	
11	Elder RFID	TinyOS	n	
12	Murphy potatoes	TinyOS	n	
13	Firewxnet	Mantis	y	
14	AlarmNet	TinyOS	n	
15	Ecuador Volcano	TinyOS	n	Deluge [121]
16	Pet game	TinyOS	n	Mate VM + TinyScript [**?**]
17	Plug	custom	y	
18	B-Live	custom	y	
19	Biomotion	custom	y	
20	AID-N	?	?	
21	Firefighting	TinyOS	n	Deluge [121]?
22	Rehabil	TinyOS	n	

. . .

Table A.12 – continued

Nr	Codename	OS used	Self-made OS	Middleware used
23	CargoNet	custom	y	
24	Fence monitor	ScatterWeb	y	FACTS [177]
25	BikeNet	TinyOS	n	
26	BriMon	TinyOS	n	
27	IP net	Contiki	n	
28	Smart home	TinyOS	n	
29	SVATS	TinyOS?	n	
30	Hitchhiker	TinyOS	n	
31	Daily morning	TinyOS	n	
32	Heritage	TinyOS	n	TeenyLIME [64]
33	AC meter	TInyOS	n	
34	Coal mine	TinyOS	n	
35	ITS	custom?	y?	
36	Underwater	custom	y	
37	PipeProbe	custom	y	
38	Badgers	Contiki	n	
39	Helens volcano	TinyOS	n	customized Deluge [121], remote procedure calls
40	Tunnels	TinyOS	n	TeenyLIME [64]

Table A.13: Deployments: software level tasks

Nr	Codename	Kernel service count	Kernel services	App-level task count	App-level tasks
1	Habitats	2	MAC, routing	1	sensing + caching to flash + data transfer
2	Minefield	?	linux services	11	
3	Battlefield	2	MAC, routing	2 + 4	Entity tracking, status, middleware (time sync, group mgmt, sentry service, dynamic conf)
4	Line in the sand	?	?	?	?
5	Counter-sniper	?	?	?	?
6	Electro-shepherd	?	-	1	sense and send
7	Virtual fences	?	MAC	1	sense and issue warning (play sound file)
8	Oil tanker	0		4	cluster formation and time sync, sensing, data transfer
9	Enemy vehicles	?	?	?	?
10	Trove game	1	MAC	3	sense and send, receive, buzz
11	Elder RFID	1	MAC	2	query RFID, report
12	Murphy potatoes	2	MAC, routing	1	sense and send
13	Firewxnet	2	MAC, routing	2	sensing and sending, reception and time-sync
14	AlarmNet	?	?	3	query processing, sensing, report sending

...

Table A.13 – continued

Nr	Codename	Kernel service count	Kernel services	App-level task count	App-level tasks
15	Ecuador Volcano	3	time sync, remote re-program, routing	3	sense, detect events, process queries
16	Pet game	2	MAC, routing	?	sense and send, receive config
17	Plug	2	MAC, routing, radio listen	2	sensing and statistics and re-port, radio RX
18	B-Live	?	?	3	sensing, actuation, data trans-fer
19	Biomotion	2	MAC, time sync	1	sense and send
20	AID-N	3	MAC, routing, trans-port	3	query processing, sensing, re-port sending
21	Firefighting	1	routing	2	sensing and sending, user input processing
22	Rehabil	0?	?	1	sense and send
23	CargoNet	0?	?	1	sense and send
24	Fence monitor	2	MAC, routing	4	sense, preprocess, report, re-ceive neighbor response
25	BikeNet	1	MAC	5	hello broadcast, neighbor dis-covery and task reception, sensing, data download, data upload
26	BriMon	3	Time sync, MAC, rout-ing	3	sensing, flash storage, sending
27	IP net	?	?	?	?
28	Smart home	?	?	?	?
29	SVATS	2	MAC, time sync	2	listen, decide
30	Hitchhiker	4	MAC, routing, trans-port, timesync	1	sense and send
31	Daily morning	1	MAC	1	sense and send
32	Heritage	?	?	?	?
33	AC meter	?	?	2	sampling, routing
34	Coal mine	2	MAC, routing	2	receive beacons, send beacon and update neighbor map and report accidents
35	ITS	2	MAC, routing	1	listen for queries and sample and process and report
36	Underwater	2	MAC, timesync	3	sensing + sending, reception, motor control
37	PipeProbe	0?	-	1	sense and send
38	Badgers	3	MAC, routing, UDP connection establish-ment	1	sense and send
39	Helens volcano	5	MAC, routing, trans-port, time sync, remote reprogram	5	sense, detect events, compress, RPC response, data report
40	Tunnels	2	MAC, routing	1	sense and send

Table A.14: Deployments: task scheduling

Nr	Codename	Time sensitive app-level tasks	Preemptive scheduling needed	Task comments
1	Habitats	0	n	sense + cache + send in every period
2	Minefield	7+	y	complicated localization, network awareness and co-operation
3	Battlefield	0	n	
4	Line in the sand	1 ?	n	
5	Counter-sniper	3?	n	localization, synchronization, blast detection
6	Electro-shepherd	?	?	
7	Virtual fences	?	n	
8	Oil tanker	1	y	user-space cluster node discovery and sync is time critical
9	Enemy vehicles	0	n	
10	Trove game	0	n	
11	Elder RFID	0	n	
12	Murphy potatoes	0	n	
13	Firewxnet	1	y	sensing can take up to 200ms, should be preemptive
14	AlarmNet	0	n	-
15	Ecuador Volcano	1	y	sensing is time-critical, but it is stopped, when query received
16	Pet game	0	n	
17	Plug	0	n	
18	B-Live	0	y	
19	Biomotion	0	y	preemption needed for time sync and TDMA MAC
20	AID-N	0	n	
21	Firefighting	0	n	
22	Rehabil	?	?	
23	CargoNet	0	n	wake up on external interrupts, process them, return to sleep mode
24	Fence monitor	0	n	if preprocess is time-consuming, preemptive scheduling needed
25	BikeNet	1	y	sensing realized as app-level TDMA schedule and is time-critical. Data upload may be time consuming, therefore preemptive scheduling may be required
26	BriMon	0	n	sending is time critical, but in MAC layer
27	IP net	0	?	
28	Smart home	?	?	
29	SVATS	0	y	preemption needed for time sync and MAC
30	Hitchhiker	0	y	preemption needed for time sync and MAC
31	Daily morning	0	n	
32	Heritage	1	y	preemptive scheduling needed for time sync?
33	AC meter	0	n	
34	Coal mine	0	n	preemptive scheduling needed, if neighbor update is time consuming
35	ITS	0	n	
36	Underwater	0	y	preemption needed for time sync and TDMA MAC
37	PipeProbe	0	n	no MAC, just send
38	Badgers	0	n	
39	Helens volcano	0	y	preemption needed for time sync and MAC
40	Tunnels	0	n	

Table A.15: Deployments: time synchronization

Nr	Codename	Time-sync used	Accuracy, μsec	Advanced time-sync	Self-made time-sync
1	Habitats	n	-	-	-
2	Minefield	y	1000	?	?
3	Battlefield	y	?	n	y
4	Line in the sand	y	110	n	y
5	Counter-sniper	y	17.2 (1.6 per hop)	y	y
6	Electro-shepherd	n	-	-	-
7	Virtual fences	n	-	-	-
8	Oil tanker	y	?	n	y
9	Enemy vehicles	n	-	-	-
10	Trove game	n	-	-	-
11	Elder RFID	n	-	-	-
12	Murphy potatoes	n	-	-	-
13	Firewxnet	y	>1000	n	y
14	AlarmNet	n	-	-	-
15	Ecuador Volcano	y	6800	y	n
16	Pet game	n	-	-	-
17	Plug	n	-	-	-
18	B-Live	n	-	-	-
19	Biomotion	y	?	n	y
20	AID-N	n	-	-	-
21	Firefighting	n	-	-	-
22	Rehabil	n	-	-	-
23	CargoNet	n	-	-	-
24	Fence monitor	n	-	-	-
25	BikeNet	y	1ms?	n, GPS	n
26	BriMon	y	180	n	y
27	IP net	?	?	?	?
28	Smart home	?	?	?	?
29	SVATS	y, not implemented	-	-	-
30	Hitchhiker	y	?	n	y
31	Daily morning	n	-	-	-
32	Heritage	y	732	y	y
33	AC meter	n	-	-	-
34	Coal mine	n	-	-	-
35	ITS	n	-	-	-
36	Underwater	y	?	?	y
37	PipeProbe	n	-	-	-
38	Badgers	n	-	-	-
39	Helens volcano	y	1ms?	n, GPS	n
40	Tunnels	n	-	-	-

Table A.16: Deployments: localization

Nr	Codename	Localization used	Localization accuracy, cm	Advanced Localization	Self-made Localization
1	Habitats	n	-	-	-
2	Minefield	y	+/-25	y	y

. . .

Table A.16 – continued

Nr	Codename	Localization used	Localization accuracy, cm	Advanced Localization	Self-made Localization
3	Battlefield	y	couple feet	n	y
4	Line in the sand	n	-	-	-
5	Counter-sniper	y	11	y	y
6	Electro-shepherd	y, GPS	>1m	n	n
7	Virtual fences	y, GPS	>1m	n	n
8	Oil tanker	n	-	-	-
9	Enemy vehicles	y	?	n	y
10	Trove game	n	-	-	-
11	Elder RFID	n	-	-	-
12	Murphy potatoes	n	-	-	-
13	Firewxnet	n	-	-	-
14	AlarmNet	y	room	n, motion sensor in rooms	y
15	Ecuador Volcano	n	-	-	-
16	Pet game	n	-	-	-
17	Plug	n	-	-	-
18	B-Live	n	-	-	-
19	Biomotion	n	-	-	-
20	AID-N	n	-	-	-
21	Firefighting	y	<5m?	n	y
22	Rehabil	n	-	-	-
23	CargoNet	n	-	-	-
24	Fence monitor	n	-	-	-
25	BikeNet	y, GPS	>1m	n	n
26	BriMon	n	-	-	-
27	IP net	n	-	-	-
28	Smart home	n	-	-	-
29	SVATS	y	?	n, RSSI	y
30	Hitchhiker	n	-	-	-
31	Daily morning	y	room	n	y
32	Heritage	n	-	-	-
33	AC meter	n	-	-	-
34	Coal mine	y	?	n, static	y
35	ITS	y, static	?	n	n
36	Underwater	y	?	n	y
37	PipeProbe	y	8cm	y	y
38	Badgers	n	-	-	-
39	Helens volcano	n	-	-	-
40	Tunnels	n	-	-	-

Table A.17: Deployments: remote access

Nr	Codename	Remote debug and data access	Remote reprogram
1	Habitats	n	n
2	Minefield	y	y
3	Battlefield	y	y

...

Table A.17 – continued

Nr	Codename	Remote debug and data access	Remote reprogram
4	Line in the sand	y	y
5	Counter-sniper	n	n
6	Electro-shepherd	?	n
7	Virtual fences	?	?
8	Oil tanker	?	n
9	Enemy vehicles	y	y
10	Trove game	n	n
11	Elder RFID	n	n
12	Murphy potatoes	n	y
13	Firewxnet	y, data	n
14	AlarmNet	y, remote queries	y, reconfigure
15	Ecuador Volcano	y	y
16	Pet game	n	y
17	Plug	n	n
18	B-Live	?	?
19	Biomotion	n	n
20	AID-N	y	y
21	Firefighting	y	y?
22	Rehabil	n	n
23	CargoNet	n	n
24	Fence monitor	n	n
25	BikeNet	n	n
26	BriMon	n	n
27	IP net	n	n
28	Smart home	?	?
29	SVATS	n	n
30	Hitchhiker	y	n
31	Daily morning	n	n
32	Heritage	?	y
33	AC meter	y	y
34	Coal mine	y, queries	n
35	ITS	y, data	n
36	Underwater	?	?
37	PipeProbe	n	n
38	Badgers	?	?
39	Helens volcano	y	y
40	Tunnels	y	y

B Hardware platform survey detailed results

Table B.1: Sensor network motes designed in years 2010 and 2011

Year	Affilation	Country	Mote name	Mote class, application
2010	UC Berkeley	USA	Egs	Medical applications
2010	Libellium	Spain	Wasp	General purpose, with sensor extenstion boards for multiple applications
2010	Oracle Sun	USA	SunSPOT	High level interface for programmers without embedded system knowledge
2010	University of Washington	USA	SNUPI	Ultra low-power home & office sensor nodes with TX only
2010	TU Braunschweig	Germany	Akiba	Rapid WSN prototyping
2010	MIT	USA	Cattle mote	Cattle gathering
2010	USDA-Agricultural Research Service	USA	Gen-II	Precision agriculture
2010	Graz University of Technology	Austria	RiverMote	River monitoring
2010	Nanjing University of Aeronautics and Astronautics	China	Structural mote	Structural monitoring
2010	Clarkson University	USA	WISAN mote	Structural monitoring
2011	University of Colima	Mexico	Agro mote	Precision agriculture
2011	Shenzhen Institute of Advanced Technology	China	BSN mote	Body sensor network

Table B.2: Sensor network motes 2010-2011: MCU and memory

Mote name	MCU name	MCU MHz	Arch, bits	RAM, KB	Program flash, KB
Egs	Atmel SAM3U2C, Cortex M3	96	32	36	128
Wasp	Atmel ATMega1281	8	8	8	128
SunSPOT	Atmel SAM9G20, ARM Thumb + Atmel ATMega168V	400 + 8	32 + 8	1024 + 1	8192 + 16
SNUPI	TI MSP430F2013, MSP430	16	16	0.128	2
Akiba	Microchip PIC18F14K22, 8051	16	8	0.512	16
Cattle mote	NXP LPC2148, ARM7	60	32	32	512
Gen-II	Microchip PIC16F883, 8051	8	8	0.256	4
RiverMote	TI MSP430F1611, MSP430	8	16	10	48
Structural mote	TI MSP430F1611, MSP430	8	16	10	48
WISAN mote	TI MSP430F1611, MSP430	8	16	10	48
Agro mote	NXP LPC2148, ARM7	60	32	32	512
BSN mote	TI MSP430F2418, MSP430	16	16	8	116

Table B.3: Sensor network motes 2010-2011: radio communication

Mote name	Radio chip	Radio frequency	Radio standard
Egs	Mitsumi WML-C46 + TI CC2520	2.4 GHz	Bluetoooth Class 2 + 802.15.4
Wasp	XBee (Zibgee and RF alternatives available)	2.4 GHz (433, 868, 915 MHz available)	802.15.4/Zigbee (FSK RF alternatives)
SunSPOT	TI CC2420	2.4 GHz	802.15.4
SNUPI	Custom	27 MHz	FSK
Akiba	TI CC2500, wake-on-radio	2.4 GHz	GFSK
Cattle mote	Aerocomm AC4790	900 MHz	FHSS
Gen-II	Maxstream XBee	2.4 GHz	802.15.4
RiverMote	MRF24J40MB	2.4 GHz	802.15.4
Structural mote	TI CC2420	2.4 GHz	802.15.4
WISAN mote	TI CC2420	2.4 GHz	802.15.4
Agro mote	Maxstream XBee Pro	2.4 GHz	802.15.4
BSN mote	Nordic nRF905	2.4 GHz	802.15.4

C MansOS

C.1 MansOS Execution models

Two application execution models are used in WSN operating systems:

- Event-based (asynchronous),
- Thread-based (synchronous).

Event-based model is simpler and requires less resources: no context switch is required between multiple simultaneously executing tasks, and single memory stack can be shared among all the tasks to save RAM. On the other hand, this model is more challenging for the programmer, especially for one who is developing lengthy applications. For event based execution, program flow is not reflected in the source code. The user has to keep the system state explicitly in custom state variables. Each event of interest is assigned a handler, that is called whenever the event occurs. Problems may arise when there coexist time-critical and time-intensive tasks in the system - the latter may cause the former to wait too long.

The benefits of thread-based model can be observed in application code, as it becomes easier to write and understand. On the other hand, this approach is not only more heavyweight, but application execution becomes more difficult to trace, stack overflow errors as well as race conditions become possible, and the complexity of OS kernel increases.

Taking all this into account, MansOS offers three scheduling models and lets the user choose between:

- Manual event handling, straight in the interrupt context.
- Preemptive multi-threading.
- Cooperative event handling, wrapped in a multi-threading syntax.

C.1.1 Event-based execution

This is the default implementation used in MansOS. In event-based execution model, the user registers and implements event handlers or *callbacks*. Sample application using event-based execution is shown in Figure C.1.

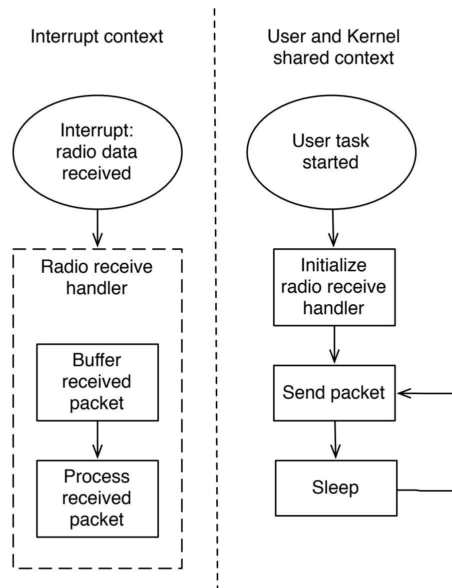

Figure C.1: Flowchart of MansOS application using event-based execution model - transmission and reception of periodic data packets

Take software timers (named *alarms* in MansOS) as an example. Alarm callback function pointers are put in a global list, ordered by alarm firing time. The list is processed in the periodic timer interrupt handler, executed 100 times per second (user-configurable value). Therefore, timers with precision up to 10 ms are available by default.

Similar callbacks can be registered for packet reception, whether serial or radio. User callbacks are executed immediately after hardware signals arrival of new data, therefore the smallest possible delay is guaranteed. However, user callback code is executed in the interrupt context and can cause problems: if the execution blocks for too long new interrupts the following can be missed; if the user code re-enables interrupts stack overflow may occur due to cascaded interrupt handling. User and kernel code is executed in the same context in the event-based model. There is no explicit task scheduler.

Energy efficiency in this model can be achieved by calling one of `sleep()` family functions in application's main loop.

C.1.2 Threaded execution

MansOS uses prioritized thread implementation, where the kernel thread has the highest priority. It is used for system event processing only and cannot be interrupted by user threads, while user threads can interrupt each other. A timer is used to implement time-sliced thread switching.

At least two threads are always created: a user thread and the kernel thread. Multiple user threads are optionally available. In the latter case, two scheduling policies are available: *round-robin*, in which the least recently run user thread is always selected, and *priority-based*, in which the thread with the highest priority is always selected (from all threads that are ready to run).

Sample application using threaded execution is shown in Figure C.2.

Mutexes are available as means of synchronization. Sequential execution of two threads can be implemented using a mutex.

Energy efficiency using threaded execution can be achieved by calling one of `sleep()` functions in the main loops of every user thread. The system will enter low power mode if no threads (including the kernel thread) are active.

Preemptive multithreading has two benefits. First, it allows users to write sequential programs without implementing split-phase state machines with external variables to store state of the machine (application). Second, it ensures equal time distribution between multiple threads and may help in scenarios, where a time-intensive is blocking time-critical thread. MansOS provides third alternative task scheduling approach using protothreads, that offers the first above mentioned benefit of sequential thinking and code execution without managing time-sliced context switch between multiple threads.

C.1.2.1 Cooperative proto-threads

Contiki protothreads approach [137] has proved itself as an effective tradeoff between usability and resource efficiency. It provides an interface that looks sequential to programmers on top of scheduler with asynchronous event handling. Primitives called *local continuations* are used to implement event handlers, called *protothreads*. Protothreads may contain statements, that exit the thread function and reenter the handler again, continuing code execution at the place where it previously left of, simulating an interface of sequential execution.

MansOS borrows protothread scheduling approach by adapting Contiki OS code. A sample application using protothreads is shown in Figure C.3

On top of protothreads, a process scheduling layer (also borrowed from Contiki OS) is used. It includes scheduling of multiple processes, storing of incoming events and

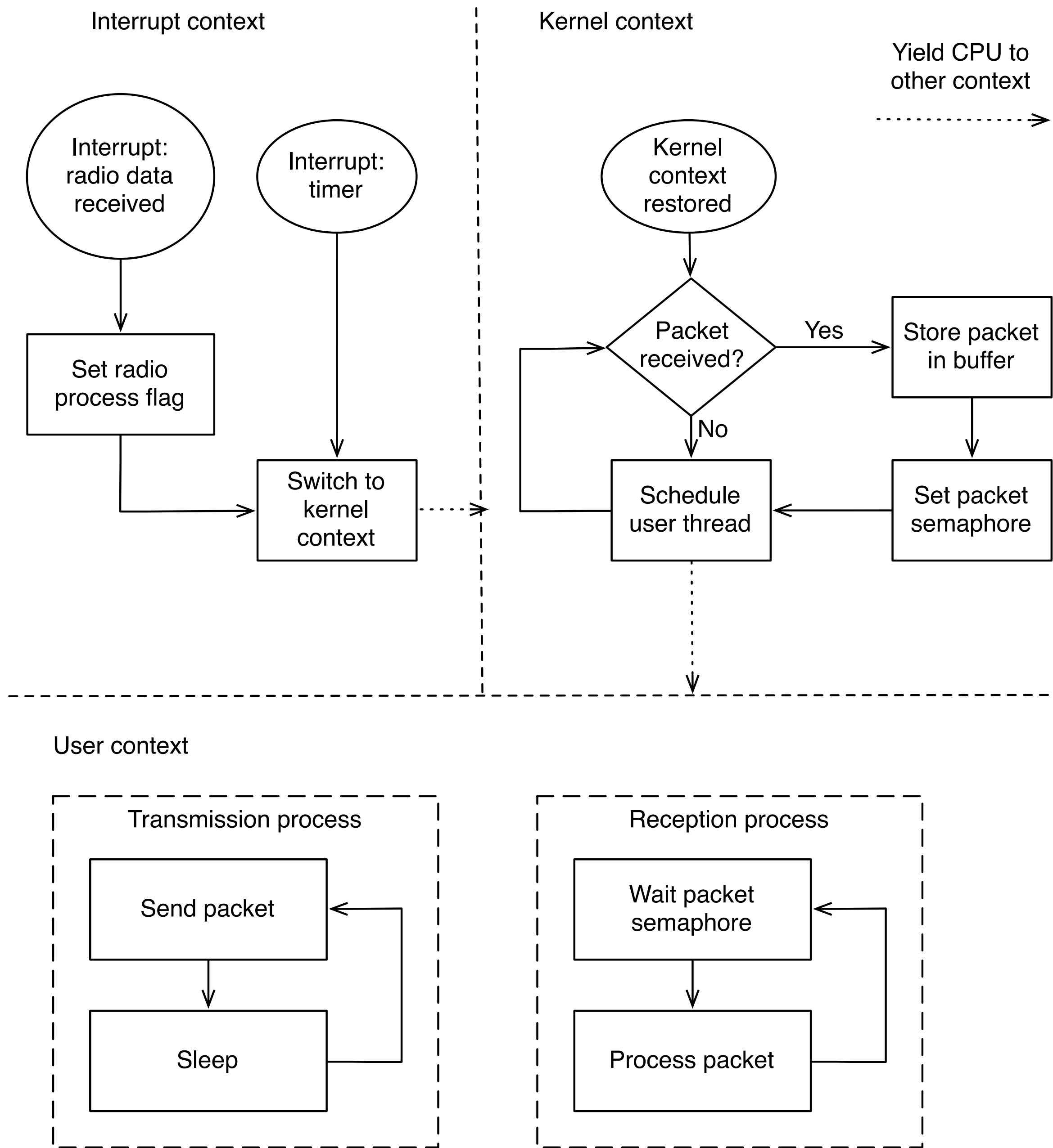

Figure C.2: Flowchart of MansOS application using preemptive threads - transmission and reception of periodic data packets

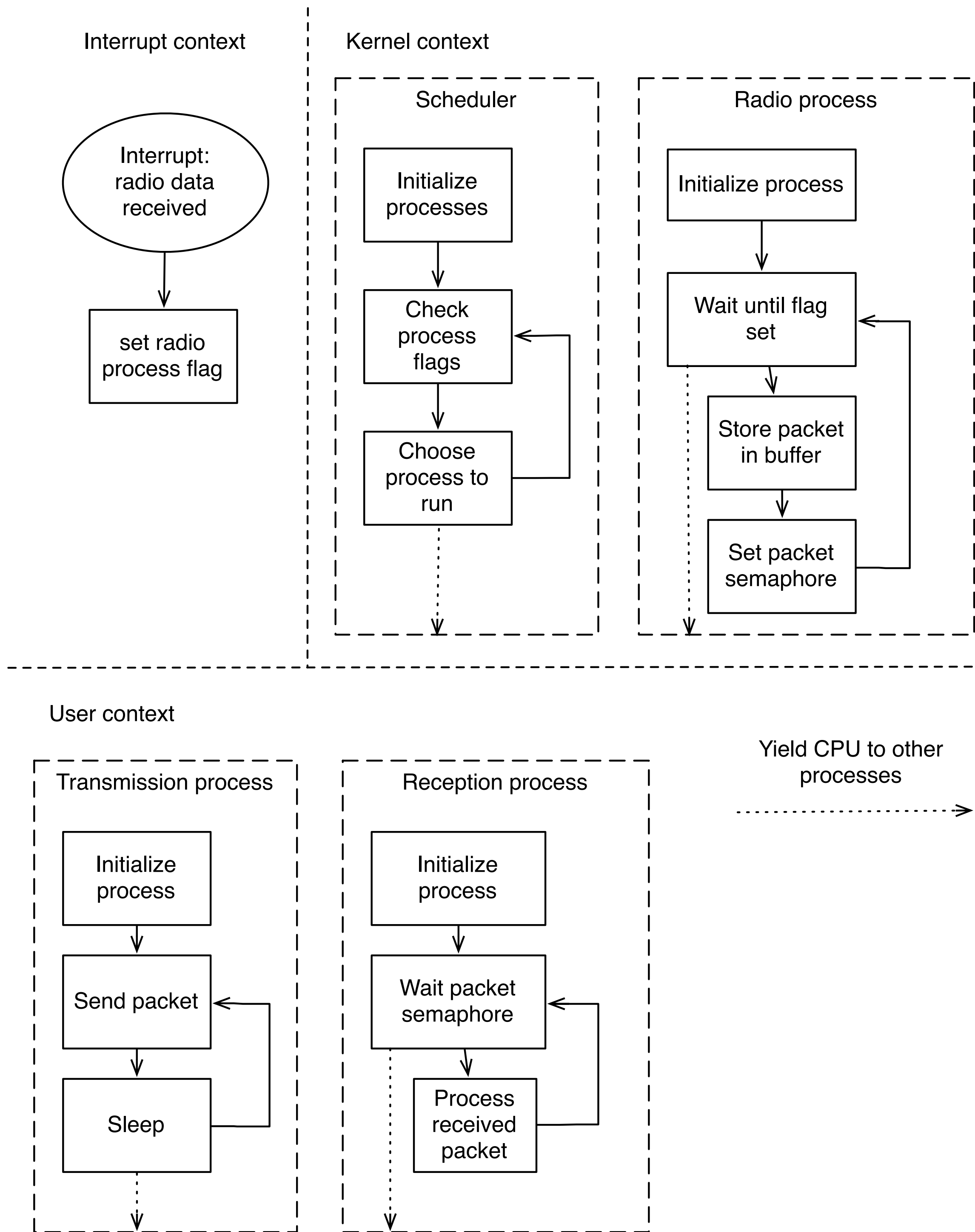

Figure C.3: Flowchart of MansOS application using cooperative proto-threads - transmission and reception of periodic data packets

passing them to processes. Inter-process communication may occur by passing an event between two processes in either synchronous or asynchronous way. In the synchronous case, the receiver process thread function is called directly. In the asynchronous case, an event is stored in the waiting event list, and it is passed as a regular event to the receiving process after all the previously received events are processed sequentially.

When using this execution model, all the processes share a common variable stack. A cooperative scheduler is started in the kernel main function. It sequentially calls all the processes that are listed in the active process list. The only mandatory process is timer handling. All other processes are created in one of two ways: either in the platform initialization code or in user application module (using specific `autostart_processes` data structure).

The benefit of using protothreads is increased efficiency - context switch occurs only at moments when a process explicitly starts to wait for an event (such as timer expiration or incoming radio packet) and passes execution to other processes. When all the processes are waiting for an event, system may switch to low-power mode until an event is received. When processes are programmed in a correct fashion, this execution model is also very effective - no unnecessary context switches are performed and time-critical event handling is ensured.

The drawbacks are mainly related to more responsibility enforced on the process developer:

- Processes must be written very carefully. Otherwise a single incorrect process may block execution of all the remaining processes. All device drivers are also implemented as protothread processes, therefore a single user-space process may block device driver execution.
- Only static variables should be used, as all the local variables stored on the stack are reset each time the process re-enters the thread handler function after the reception of an event. This requirement may introduce new software bugs, as the usage of local variables is legal from syntax perspective and the compiler issues no warnings.
- Processes may call statements that wait for an event or simply yield control to other processes only from the main process thread function (due to technical protothread implementation details [137]).
- Programmers must include predefined keywords at the beginning (`PROCESS_BEGIN()`) and end (`PROCESS_END()`) of the process thread function. These macro-keywords ensure some *behind the scenes* protothread technical requirements are met.

C.2 MansOS networking protocol stack

MansOS is a flexible WSN operating system, that allows users to access radio communication at multiple ISO Open Systems Interconnection (OSI) networking stack layers [52]: physical, data link (MAC), and network. The following subsections describe each of these layers.

C.2.1 Physical layer

The role of physical layer (PHY) is exchange of data bits using physical communication link. In WSN case the communication link is wireless. Today commercial radio communication chips offer rich feature set, far beyond simple bit transmission. For example, TI CC2420 radio chip also solves the framing problem (a link layer problem) by providing data transmission and reception in whole packets. Data encryption using AES-128 [178] is also available as an additional feature.

In PHY layer MansOS provides function `radioSend(data, len)` for data transmission. Reception in PHY layer is possible either using event-driven execution model and handling callback function (set by `radioSetReceiveHandle(callback)`) directly in interrupt context, or using cooperative execution model with blocking `waitRadioPacket()` function.

The advantage of using directly PHY layer is high performance, that comes with a price of user responsibility to write error-free data reception handlers and customized networking logic.

C.2.2 MAC layer

In WSN context data link layer is usually reduced to MAC layer responsibility - choosing of appropriate moments, when to send data. Encryption and framing is often already solved in the PHY layer. Irregularities due to harsh environment and wireless low-power communication specifics introduce many problems in sensor networks. Therefore optimizations in MAC layer are far more important than in the global internet. Another reason for the variety of MAC protocols used in WSNs is the lack of standardization. IPv6 and its low-power implementation, 6lowpan, introduce restrictions in this context.

To provide interchangeability, MAC protocols are defined as data structure `MacProtocol_t`. It stores pointers to protocol functions `init()`, `send()`, `poll()`, and several other internal routines.

Similarly to PHY layer, also in MAC layer MansOS provides data transmission and reception callback functions. Nevertheless, in contrast to PHY, these functions are

executed outside interrupt context. Kernel handles packet reception and data buffering. MAC protocol's `poll()` function is called in kerne thread context. User level data transmission function is `macSend(dstAddress, buffer, bufferSize)`. Data reception handler setup: `macProtocol.recvCb = radioDataHandler`.

C.2.2.1 Network layer

MansOS network layer consists of two parts: routing and socket interface.

In networking layer MansOS implements Unix-like socket interface. Socket is opened using `socketOpen(Socket_t *s, SocketRecvFunction *callback)`, bound to a particular port using `socketBind(Socket_t *socket, int port)`. Data is sent using `socketSetDstAddress(Socket_t *s, MosShortAddr addr)` to set destination node address and `socketSend(Socket_t *s, void *data, int len)` to execute the actual data sending. And finally socket can be closed using `socketClose(Socket_t *s)`. Data reception callback function is initialized in `socketOpen()` function and is called in kernel context. Example application is shown in Listing C.1.

Listing C.1: MansOS socket application example - data is sent to base station, using port 123, and received packet length is printed

```
#include "stdmansos.h"
#include <net/socket.h>

enum { DATA_PORT = 123, SLEEP_TIME_MS = 2000 };

static void recvData(Socket_t *socket, uint8_t *data, uint16_t len) {
    PRINTF("got \%d bytes from 0x\%04x\n",
            len, socket->recvMacInfo->originalSrc.shortAddr);
    redLedToggle();
}

void appMain(void) {
    Socket_t socket;
    socketOpen(&socket, recvData);
    socketBind(&socket, DATA_PORT);
    socketSetDstAddress(&socket, MOS_ADDR_ROOT);

    uint16_t counter;
    for (;;) {
        if (socketSend(&socket, &counter, sizeof(counter))) {
            PRINT("socketSend failed\n");
        }
        ++counter;
        mdelay(SLEEP_TIME_MS);
    }
}
```

Routing allows data packets to be sent in multi-hop networks. MansOS interface for routing protocol design is simple. To add a new routing protocol, only two functions must be implemented: `initRouting` un `routePacket()`. Function `routePacket()` decides what to do next with each received packet, and returns the decision as function return value. MansOS platform-independent communication layer performs actions according to routing protocol decision - either forwards or drops the packet.

D OOMOS

D.1 Object-oriented operating system advantages

This section summarizes advantages of creating object-oriented operating systems, as described by Vincent Frank Russo, in 1991, in his PhD thesis on the creation of the Choices OS [162]:

- Portability. Abstract classes is a convenient way to specify what functions are required to develop a new platform or device driver.

- Code reuse. Improved by inheritance. New platforms may extend base classes, override only required functions.

- Separation of policies from mechanisms. By creating hierarchies of classes and requiring a particular non-leaf class as an interface, subclasses may change some mechanisms, leaving the global policies intact.

- Optimizations by subclassing. Some algorithms may be implemented in multiple ways. By changing the implementing class we can choose (even in runtime) which dimension to optimize.

- Portability/Efficiency tradeoff. A more portable class may be used at start and replaced by more efficient, yet more platform-specific, class later. For example, bzero and bcopy functions of a MemoryBlock class may use platform-independent loop at start and be replace by platform-specific instructions later.

- Easier unit-testing. Simple tests can be created by using interfaces provided by objects and stub-implementations of member functions.

- Simpler synchronization and mutual exclusion. Each object holds its semaphores, mutexes and other synchronization primitives inside it.

It is also noted, that the overhead imposed by C++ language is reasonable, and efficient-enough implementations are possible, as it is shown in a research paper summarizing Choices OS performance evaluation results [163].

D.2 OOMOS source code examples

Listing D.1: OOMOS interface example - `ILogStream` interface for data logging to a stream, provides pure virtual functions `logStreamOpen()`, `logStreamClose()`, and `logStreamWrite()`, which must be implemented by classes providing `ILogStream` interface

```
#include <iface/iface.h>

class ILogStream {
public:
    virtual err_t logStreamOpen() = 0;
    virtual err_t logStreamClose() = 0;
    virtual uint16_t logStreamWrite(const void *buf, uint16_t len) = 0;
};
```

Listing D.2: OOMOS UART interface - abstract class that declares functions required for objects implementing `IUART` interface, and also implements `ILogStream` interface

```
#include <iface/ilogstream.h>
#include <eventhandler.h>

class IUART : public ILogStream {
public:
    //--------------------------------------------------------------
    // IUART interface declaration
    //--------------------------------------------------------------
    virtual err_t initUart(uint32_t speed) = 0;
    virtual err_t uartEnableTx() = 0;
    virtual err_t uartDisableTx() = 0;
    virtual void uartTxByte(uint8_t b) = 0;
    virtual err_t uartEnableRx() = 0;
    virtual err_t uartDisableRx() = 0;
    virtual err_t uartSetRxHandler(EventHandler *handler) = 0;

    //--------------------------------------------------------------
    // ILogStream interface implementation
    //--------------------------------------------------------------
    virtual err_t logStreamOpen() { return uartEnableTx(); }
    virtual err_t logStreamClose() { return uartDisableTx(); }
    virtual uint16_t logStreamWrite(const void *buf, uint16_t len);
};
```

Listing D.3: OOMOS radio interface - abstract class that declares functions required for objects implementing `IRadio` interface, and also implements `ILogStream` interface

```
#include "iface/ilogstream.h"
#include <eventhandler.h>

class IRadio : public ILogStream {
    EventHandler *eventHandler;
public:
    //---------------------------------------------------------------------
    // IRadio interface declaration
    //---------------------------------------------------------------------
    virtual err_t init() = 0 ;
    virtual err_t send(const void *data, uint16_t len) = 0;
    virtual err_t enableRx() = 0;
    virtual void disableRx() = 0;
    virtual int16_t packetRecv(void *buf, uint16_t len) = 0;

    // Received data handling
    void setReceiveHandler(EventHandler *eh) { eventHandler = eh; }
    void onPacketRx() {
        if (eventHandler) eventHandler->handleEvent(E_RADIO_RX);
    }

    //---------------------------------------------------------------------
    // ILogStream interface implementation
    //---------------------------------------------------------------------
    virtual err_t logStreamOpen() { return ERR_OK; }
    virtual err_t logStreamClose() { return ERR_OK; }
    virtual uint16_t logStreamWrite(const void *buf, uint16_t len) {
        return send(buf, len);
    }
};
```

Listing D.4: OOMOS CC2420 device driver (partial) - class acts as generic radio chip by providing `IRadio` interface, and handles GPIO pin interrupts by provided `IEventHandler` interface. Uses `ISPI`, `IMCU`, and `IGPIO` interfaces.

```
#include <iface/iradio.h>
#include <eventhandler.h>
#include <iface/ispi.h>
#include <iface/imcu.h>
#include <iface/igpio.h>

class CC2420 : public IRadio, public EventHandler {
    ISPI *spi;
    IMCU *mcu;
    IGPIO *gpio;
public:
    //---------------------------------------------------------------------
    // IRadio interface
    //---------------------------------------------------------------------
    virtual err_t init();      // Initialize radio
    virtual err_t enableRx(); // Enable reception
    virtual void disableRx(); // Disable reception
    virtual err_t send(const void *data, uint16_t len);
    // Store received packet in buffer
    virtual int16_t packetRecv(void *buf, uint16_t len);
```

```
    //------------------------------------------------------------------------
    // EventHandler interface
    //------------------------------------------------------------------------
    // GPIO interrupt handling callback
    virtual err_t handleEvent(Event_t event, void *data = NULL,
            uint16_t dataLen = 0); // Handle RX packet event
};
```

Listing D.5: OOMOS TelosB platform initialization (partial) - interface providers are defined and wired using standard C++ syntax, without additional configuration files (present in TinyOS)

```
err_t Telosb::init() {
    // ...
    // bind MCU GPIO and LED components
    leds.setGpio(mcu.getGPIO());
    leds.setCount(3); // 3 leds on TelosB: red, green, blue
    leds.setLedPin(0, LEDS_RED_PORT, LEDS_RED_PIN, LEDS_ON_HIGH);
    leds.setLedPin(1, LEDS_GREEN_PORT, LEDS_GREEN_PIN, LEDS_ON_HIGH);
    leds.setLedPin(2, LEDS_BLUE_PORT, LEDS_BLUE_PIN, LEDS_ON_HIGH);
    // Initialize LED component
    err_t e = leds.init();
    if (e != ERR_OK) return e;
    // ...
    // Initialize radio module
    radio.setMcu(&mcu);
    radio.setSpi(mcu.getSpi(RADIO_SPI_ID));
    e = radio.init();
    if (e != ERR_OK) return e;
    // ...
    return ERR_OK;
}
```

Listing D.6: OOMOS interface for abstract hardware platform - declares mandatory API as pure-virtual functions and optional API as virtual functions with default (empty) implementation

```
#include <iface/imcu.h>
#include <iface/ileds.h>
#include <iface/iadc.h>
#include <iface/iradio.h>
#include <iface/istorage.h>

class IPlatform {
public:

    // ------------------------------------------------------------------------
    // Mandatory API
    // ------------------------------------------------------------------------
    virtual err_t init() = 0;
    // Delays
    virtual void udelay(uint16_t us) = 0; // in microseconds
```

```
    virtual void mdelay(uint16_t ms) = 0; // in milliseconds
    virtual void delay(uint16_t s) = 0;   // in seconds
    virtual uint32_t getUptimeJiffies(); // uptime in MCU jiffies
    virtual uint32_t getUptimeMs() = 0;  // uptime, in milliseconds
    // Run a callback function after jf clock jiffies
    virtual err_t scheduleTimer(jiffie_t jf, TimerCallback_t handler) = 0;
     // Return sleep, if the platform can switch to sleep mode at the moment
    virtual bool canSleep() { return true; }
    virtual void enterSleepMode() = 0;

    // ---------------------------------------------------------------
    // Common, yet optional API
    // ---------------------------------------------------------------
    virtual IMCU *getMCU() { return 0; } // PC may not have it
    virtual IADC *getAdcModule() { return 0; }
    virtual IRadio *getRadio() { return 0; }
    virtual ILEDs *getLEDs() { return 0; }
    virtual IUART *getPrintUart() { return 0; }
    virtual IStorage *getStorage() { return 0; }
    // Sensors
    virtual uint16_t readLight() { return 0; }
    virtual uint16_t readInternalTemp() { return 0; }
    virtual uint16_t readTemperature() { return 0; }
    virtual uint16_t readHumidity() { return 0; }
    virtual uint16_t readAccelX() { return 0; }
    virtual uint16_t readAccelY() { return 0; }
    virtual uint16_t readAccelZ() { return 0; }
};
```

Listing D.7: OOMOS protocol interface and base class prototypes -

```

#include <iface/iface.h>

class IProtocol {

public:
    /**
     * Initialize protocol. Run this method AFTER setting all subcomponents!
     */
    virtual err_t init() = 0;

    /**
     * Initialize protocol relations, set protocols that
     * will be right above/under this one
     */
    virtual void setRelations(IProtocol *topProto, IProtocol *bottomProto) = 0;

    /**
     * Return header size (in bytes) that this protocol will add to packet buffer
     */
    virtual uint8_t getHeaderSize() = 0;

    /**
     * Process a packet and push it up in the networking protocol stack
     * (forward it to the next protocol right above this one)
```

```
     */
27   virtual void pushUp(PacketBuffer *packet) = 0;

29   /**
      * Process a packet and pull it down in the networking protocol stack
31    * (forward it to the next protocol right under this one)
      */
33   virtual void pullDown(PacketBuffer *packet) = 0;
   };
35
   /**
37  * Abstract base class for protocols. Implements relations with top and
    * bottom protocols
39  */
   class AbstractProtocol : public IProtocol {
41 protected:
       IProtocol *topProto, *bottomProto;
43 public:
       virtual void setRelations(IProtocol *tp, IProtocol *bp) {
45         this->topProto = tp;
           this->bottomProto = bp;
47     }
   };
```