Double-Anchored Software Architecture for Wireless Sensor Networks

Double-Anchored Software Architecture for Wireless Sensor Networks

vorgelegt von

Vlado Handziski
(M.Sc. in Electrical Engineering)

von der Fakultät IV
Elektrotechnik und Informatik
der Technischen Universität Berlin
zur Erlangung des akademischen Grades

Doktor der Ingenieurwissenschaften
- Dr.Ing. -

genehmigte Dissertation

Tag der wissenschaftlichen Aussprache: 07.03.2011

Berlin 2011
D 83

To Ani, Elena and Jan
with all my love!

Contents

List of Figures

List of Tables

List of Acronyms

6LoWPAN	IPv6 over Low power WPAN
AAA	Authentication, Authorization and Accounting
ADAGE	Avionics Domain Application Generation Environment
ADC	Analog-to-Digital Converter
AMQP	Advanced Message Queuing Protocol
API	Application Programming Interface
ARPA	Advanced Research Projects Agency
ASEC	Attribute Service Extension Component
ASIC	Application-Specific Integrated Circuit
ASK	Amplitude-Shift Keying
AUTOSAR	Automotive Open System Architecture
CBPS	Content-Based Publish/Subscribe
CCA	Clear Channel Assessment
CCR	Capture/Compare Register
CDF	Cumulative Distribution Function
CMOS	Complementary Metal Oxide Semiconductor
CONET	Cooperating Objects Network of Excellence
CORBA	Common Object Request Broker Architecture
COTS	Commercial Off The Shelf
CPU	Central Processing Unit
CSEC	Communication Service Extension Component
CSMA	Carrier Sense Multiple Access

CSS	Chirp Spread Spectrum
CTFS	CONET Testbed Federation Server
CTF	CONET Testbed Federation
CTP	Collection Tree Protocol
DAC	Digital to Analog Converter
DASA	Double-Anchored Software Architecture
DCS	Data Centric Storage
DC	Direct Current
DECT	Digital Enhanced Cordless Telecommunications
DHCP	Dynamic Host Configuration Protocol
DHT	Distributed Hash Table
DMA	Direct Memory Access
DNS	Domain Network System
DSP	Digital Signal Processing
DSSA	Domain-Specific Software Architecture
DSSS	Direct-Sequence Spread Spectrum
DSUWB	Direct-Sequence Ultra-Wide Band
ECG	Electrocardiogram
EYES	Energy Efficient Wireless Sensor Networks
FHSS	Frequency-Hopping Spread Spectrum
FIFO	First In, First-Out
FIRE	Future Internet Research & Experimentation
FSK	Frequency-Shift Keying
FSM	Finite-State Machine
GAE	Google App Engine
GCC	GNU C Compiler
GENI	Global Environment for Network Innovations
GHT	Geographic Hash Table
GNU	GNU's Not Unix!
GPIO	General Purpose Input/Output
GPS	Global Positioning System

GUI	Graphical User Interface
HAL	Hardware Adaptation Layer
HDLC	High-Level Data Link Control
HIL	Hardware Interface Layer
HPL	Hardware Presentation Layer
HTTP	Hypertext Transport Protocol
I/O	Input/Output
I^2C	Inter-IC
IC	Integrated Circuit
ID	Identification
IEEE	Institute of Electrical and Electronics Engineers
IETF	Internet Engineering Task Force
IPC	Inter-Process Communication
IP	Internet Protocol
IR	Infrared Radiation
ISM	Industrial, Scientific and Medical
ISO	International Organization for Standardization
ISR	Interrupt Service Routine
JSON	JavaScript Object Notation
JTAG	Joint Test Action Group
JVM	Java Virtual Machine
LAN	Local Area Network
LCD	Least Common Denominator
LED	Light Emitting Diode
LOESS	Locally Weighted Scatterplot Smoothing
LQI	Link Quality Indicator
MAC	Medium Access Control
MCU	Micro-Controller Unit
MEMS	Microelectromechanical systems
MIME	Multipurpose Internet Mail Extensions
MOM	Message-oriented Middleware

NFS Network File System

NLA Network Layer Architecture

NSLU2 Network Storage Link for USB 2.0 Disk Drives

NTP Network Time Protocol

O-QPSK Offset Quadrature Phase-Shift Keying

OO Object Oriented

OSI Open Systems Interconnection Initiative

OS Operating System

OTA Over-The-Air Reprogramming

P2P Peer-to-Peer

PC Personal Computer

PHY Physical Layer

POSIX Portable Operating System Interface

PPP Point-to-Point Protocol

PWM Pulse-Width Modulation

PoE Power over Ethernet

RAM Random-Access Memory

REST Representational State Transfer

RF Radio Frequency

RISC Reduced Instruction Set Computer

RNG Random Number Generator

ROLL Routing Over Low power and Lossy networks

RPC Remote Procedure Call

RRD Round Robing Database

RS-232 Recommended Standard 232

RSSI Receive Signal Strength Indicator

RX Receive

SEC Service Extension Component

SFD Start Frame Delimiter

SLOC Source Lines of Code

SNMP Simple Network Management Protocol

SOA	Service Oriented Architecture
SPI	Serial Peripheral Interface
SP	Sensornet Protocol
SQL	Structured Query Language
SSH	Secure Shell
SUT	System Under Test
SoC	System-on-Chip
TA API	Testbed Adaptation API
TCL	Tool Command Language
TCP	Transmission Control Protocol
TEP	TinyOS Enhancement Proposal
TF API	Testbed Federation API
TKN	Telecommunication Networks Group
TWIST	TKN Wireless Indoor Sensor Network Testbed
TX	Transmit
UART	Universal Asynchronous Receiver Transmitter
UDT	User Defined Type
URI	Universal Resource Identifier
URL	Universal Resource Locator
URN	Universal Resource Name
USART	Universal Synchronous Asynchronous Receiver Transmitter
USB	Universal Serial Bus
UUID	Universally Unique Identifier
WLAN	Wireless Local Area Network
WSN	Wireless Sensor Network
WiSeNTs	Cooperating Embedded Systems for Exploration and Control with WSNs
XML	Extensible Markup Language
XMPP	Extensible Messaging and Presence Protocol
cURL	Client URL Request Library

Zusammenfassung

Nach einem Jahrzehnt intensiver Forschung und Entwicklung sind drahtlose Sensornetze (Wireless Sensor Networks (WSNs)) kurz davor, sich von einer unbekannten Technologie in ein tragfähiges Marktsegment zu verwandeln. Während dieses Zeitraums hat sich die WSN-Knoten-Hardware ständig verbessert, was zu erhöhter Funktionalität und einer Reduzierung von Formfaktor, Kosten und Energieverbrauch geführt hat. Leider konnte die Software-Entwicklung nicht mit demselben Tempo voranschreiten. Die Begrenztheit der Betriebsmittel und anwendungsspezifische Anforderungen sind dafür verantwortlich, dass Entwickler geschlossene und integrierte Lösungen anstreben, was die Wiederverwendung von Entwürfen und Programmiercode behindern. Dies führt dazu, dass der erforderliche Aufwand für die Entwicklung neuer Anwendungen und ihre Anpassung an die sich kontinuierlich entwickelnde Hardware ansteigt.

Das Fehlen einer allgemeinen Softwarearchitektur für WSNs wird von vielen Mitgliedern der wissenschaftlichen Gemeinschaft als wesentlicher Faktor für die existierenden Defizite angesehen. Wir präsentieren eine *doppelverankerte Softwarearchitektur für drahtlose Sensornetze*, die eine effektive WSN Entwicklung ermöglicht, indem traditionelle Methoden des Entwurfes und der Wiederverwendung von Programmiercode angewendet werden, unter Einhaltung von bewährten Prinzipien wie funktioneller Entkoppelung und dem Verbergen von Komplexität. Gleichzeitig wird ein Mechanismus zur Steuerung des inhärenten Kompromisses zwischen Effizienz und Wiederverwendung zur Verfügung gestellt, so dass sich die genannten Vorteile im Vergleich mit einer maßgeschneiderten und vertikal integrierten Lösung ohne einen übermäßig hohen Nachteil an Leistungsfähighkeit realisieren lassen.

Die vorgestellte Architektur ist als ein Komponenten-System ausgeführt, das um zwei "Anker" angeordnet ist, die Beständigkeit ermöglichen und als die Basis für den Entwurf und die Wiederverwendung von Programmiercode dienen. Der untere *Portabilitäts-Anker* abstrahiert die Hardware und ermöglicht explizite Kontrolle des Performance-Portabilität Trade-offs. Der obere *Interoperabilität-Anker* abstrahiert die Knoten-lokalen Dienste mithilfe einer expressiven Publish/Subscribe Schnittstelle und unterstützt anwendungsspezifische Anpassung. Diese Dissertation vertritt

die These, dass eine breite Softwarearchitektur, die auf diesen beiden Ebenen des Software-Stacks verankert ist, wirksam Portabilität und Interoperabilität fördern kann und dass dies unter Beachtung der Kosten für die involvierten Abstraktionen geschieht.

Wir werten diese Behauptungen in qualitativer und quantitativer Art und Weise aus, und zwar anhand eines Beispiels, das eine Reihe von Prototyp-Implementierungen realisiert, von denen einige breite Anwendung in der WSN-Forschungs-Gemeinschaft gefunden haben. Zur Unterstützung der Auswertung haben wir eine spezifische Test-Infrastruktur entwickelt, die eine effiziente Prüfung der funktionalen und nicht-funktionalen Eigenschaften von WSN-Protokollen und -Diensten ermöglicht.

Abstract

After a decade of intense research and development, Wireless Sensor Networks (WSNs) are on the verge of transforming from an obscure technology into a viable market segment. In this period, the WSN node hardware has constantly improved, resulting in better functionality while size, cost and energy consumption have been reduced. Unfortunately, the software development process has not been able to keep the same pace. The tight resource constraints and the application-specific requirements are driving developers into closed and integrated solutions which impede design and code reuse, increasing the required effort for developing new applications and adapting them to an ever evolving hardware.

The lack of a common software architecture for WSNs is seen by many in the research community as significant contributing factor for the existing inefficiencies. We introduce a *Double-Anchored Software Architecture* that enables effective WSN development through traditional methodologies of design and code reuse, using time-tested principles like functional decoupling and complexity hiding. At the same time, it provides mechanisms for controlling the inherent trade-offs between efficiency and reuse so the above benefits can be achieved without paying a prohibitively high price in performance, compared to a customized and vertically integrated solution.

The proposed architecture is specified in the form of a component framework organized around two "anchors" that provide rigidity and establish a base for design and code reuse. The lower *portability anchor* abstracts the hardware while enabling explicit control over the performance-portability trade-offs. The upper *interoperability anchor* abstracts the node-local services behind an expressive publish/subscribe interface and supports application-specific customization. This dissertation contends that a broad software architecture, anchored at these two levels of the software stack, can effectively promote portability and interoperability while maintaining high sensibility towards abstraction costs. We evaluate these claims in qualitative and quantitative way, on the example of several prototype implementations, some of which are in wide use in the WSN community. To support the evaluation we have also developed a custom distributed testing infrastructure that enables efficient testing of functional and non-functional properties of WSN protocols and services.

Acknowledgements

First and foremost, I like to express my deepest gratitude to my adviser, *Prof. Adam Wolisz*, for welcoming me in the Telecommunication Networks Group, for providing me with the support and guidance that made this work possible and for helping me grow as a professional and as a person. I also like to thank *Prof. Kurt Rothermel* for accepting to review the dissertation.

The work in this dissertation has been supported in part through the European Commission projects *EYES*, *WiSeNTs* and *CONET*. Many of the presented ideas were born through the interaction with members of the Wireless Embedded Systems group at UC Berkeley, lead by *Prof. David Culler*. With *Joe Polastre* we cooperated on the initial porting of the Texas Instruments MSP430 microcontrollers in TinyOS 1.x, and on generalizing the learned lessons in the form of a Hardware Abstraction Architecture subsequently used in TinyOS 2.x. This initial design was further refined and extended through the activities of the TinyOS 2.x Core Working Group, especially through my cooperation with *Kevin Klues*, *David Gay* and *Philip Levis*.

Two of my colleagues at TKN were instrumental in the realization of the presented solutions. With *Jan-Hinrich Hauer* we collaborated on the design and the implementation of the Hardware Abstraction Architecture in TinyOS 2.x and on the design and implementation of TinyCOPS. With *Andres Köpke*, we worked on the software support for the eyesIFX nodes, on the design of the TWIST testing architecture, and on the construction of the instance at the TKN office building.

My days at TKN and in my new home country would certainly not have been the same without the kind words of support of my colleagues *Petra Hutt* and *Andreas Willig*. I am deeply grateful to have them as my friends.

Finally, I wish to thank my wife for all the sacrifices throughout the years of hard work and my kids for the hope that they inspire in me. My endless gratitude goes to my parents and to my brother and sister, for their relentless support, trust and encouragement, on the winding path through life.

CHAPTER 1

Introduction

Driven by pervasive availability of affordable high-bandwidth access at the edge, and large increase in computational and storage capabilities at the core, the usage model of the Internet is experiencing a profound transformation. User generated content is becoming an important pillar on which novel services like media sharing, social networks and life-streaming are being built. The characteristic traffic asymmetry is slowly disappearing, as data increasingly flows from the edge of the network towards the core. This process is further amplified by the rapid spread of the mobile phone and its transformation from a voice into a general-purpose data capturing and communication platform. The ubiquity of wireless access and geolocation has enabled nomadic and contextual [111] generation of content. As a result, the coupling between the physical and the virtual world is becoming stronger, and more and more of the tangible things are getting their virtual duals.

Despite these developments, the bulk of the contextual information about the world around us remains human-generated and explicit, thus limited in scope and depth [193]. Vast parts of our reality remain under-instrumented and can't be easily digitized, analyzed and ultimately controlled. On the confluence of these technological and social trends a new platform is emerging that promises to fill this gap: the so-called Wireless Sensor Networks (WSNs), networks of small devices that integrate low-power sensing and processing with short-range wireless transmission [4].

1.1 Wireless Sensor Networks

The untethered nature, the small size and the need for long-lived and unattended deployments means that the nodes in a WSN often have to operate on battery power. The energy provided by the batteries can not be easily replenished during the application lifetime, leading to a limited energy budget that determines the achievable system lifetime. This energy scarcity promotes the energy-efficiency of the hardware and the software as the primary design objective [172] for this class of systems.

Due to the cost, size and energy constraints, the WSN nodes usually have a processing element in the form of a low-power 8-bit or 16-bit Micro-Controller Unit (MCU). The MCUs have small amounts of program and data memory which stands in contrast to the relatively high data buffering needs in the sensing and communication stacks. This memory-limited nature necessitates different optimization goals for the software development process than in traditional systems: the size-efficiency and the memory-efficiency of the code are often much more important than the computational-efficiency.

The sensing elements are responsible for gathering contextual information from the physical reality around the WSN node. Although the nature of the collected information and the type of the used sensors strongly depends on the application, due to cost and size pressures, many of the sensors have reduced fidelity that has to be compensated at the system level by oversampling in the spatial and temporal domains.

The same size and energy constraints also put limits on the communication subsystem. They mandate the use of small, low-cost and low-power radio transceivers. The WSN traffic is mostly comprised out of small data packets sent over relatively short distances and with low average data rates. Under these conditions, the power consumption of the transceiver is dominated by the radio electronics and protocol solutions that limit the amount of idle listening are needed to achieve long system lifetimes.

Despite this simple hardware architecture of the individual nodes, the WSNs enable construction of large distributed monitoring and actuation systems that are deeply embedded in the physical environment and offer unprecedented levels of temporal and spatial sampling density. Thanks to their small size, low cost and reduced installation and maintenance overheads, they pave the way for novel applications that were either impractical or impossible with legacy technologies. This flexibility and versatility of WSNs, however, comes at the expense of a more complex overall system design that is necessary to address the severe resource scarcity and the challenging operating conditions. The system designers commonly have to resort to redundant deployment, error-mitigation and error-correction strategies in order to construct a reliable aggregate system out of the unreliable individual building blocks.

1.2 Software Design Challenges

After a decade of intense research and development, WSNs are on the verge of transforming from an obscure technology into a viable market segment. WSNs are currently being applied in areas as diverse as environmental monitoring, building automation, industrial monitoring and control, logistics, agriculture and health-care [6]. During this period, the improvements in the silicon production process, following Moore's

Law, has driven the design of the WSN node hardware forward, resulting in better functionality while reducing the size, energy consumption and cost. This trend of continuous advancement of the WSN hardware will continue in the foreseeable future and will result in more affordable, capable and efficient components. As a result, the WSN node costs will soon be approaching the thresholds enabling massive market penetration.

Unfortunately, the software development process has not been able to keep the same pace as the hardware improvements. The tight resource constraints and the application-specific requirements typical for WSNs are driving developers into closed and integrated solutions. Although they provide high levels of efficiency by extracting maximal performance out of the available resources, the tight vertical integration impedes design and code reuse, significantly increasing the required effort for developing new applications or for adapting existing applications to an ever evolving hardware [117].

This integrated software development model is becoming a significant hindrance to faster growth of the WSN technology and is opening a productivity gap between the hardware and the software domains. Developing applications is hard and a successful outcome requires expertise in the complete value chain starting from the hardware platform, communication protocols, sensing stack, up to the application domain. These shortcomings are reflected in the current structure of the WSN market, characterized by high fragmentation and low levels of horizontal reuse. On one side, it is dominated by vertically integrated, single-vendor solutions that are focused on specific application areas. On the other side, there is a large offering of generic system components that are developed in isolation and can not be easily combined into reliable end-user solutions.

The currently predominant software development approach is ill-suited to handle the new set of challenges that the broad adoption of the WSN technology brings. In contrast to the isolated nature of classical embedded systems, WSN are networked systems that are often interconnected with the wider Internet. In the near future, a typical home will host several WSN installations providing energy metering, building automation, health-care and many other services. The physical proximity and the privacy implications of the technology will drive changes in the traditional relationship between the users and the operators of these WSN services. Users will request more direct control and freedom to compose the services in novel ways, mirroring the "content mashup" trend on the Web. Enabling such federated applications requires breaking the existing vertical silos and transforming WSNs from closed, application-specific solutions into open platforms that facilitate rapid application development and offer a degree of separation between the services and the underlying heterogeneous hardware substrate.

The lack of a common *software architecture* for WSNs is seen by many in the research community as significant contributing factor for the existing inefficiencies [16, 80, 177, 192]. The IEEE Standard 1471 defines software architecture as *"the fundamental organization of a system embodied in its components, their relationships to each* *other, and to the environment, and the principles guiding its design and evolution"* [103]. Through the identification of core components, their interfaces and composability rules, software architectures codify the functional decomposition of the system and lay foundations for a more structured development approach. In addition, they formalize best-practices into reusable design templates that can be shared across different applications and hardware platforms [7]. The resulting increase in adaptability and reusability reduces the cost and the complexity of the development process and lowers the entry barriers for new developers and new application areas.

Although there is a broad agreement on the benefits of converging towards a more unified software architecture for WSNs, the question about the appropriate scope and granularity of such an architecture remains open. Most of the existing proposals concentrate on selected subsets of the WSN software stack, with great diversity in the context in which the proposals are framed: execution environments, communication architectures, programming and service abstractions, middleware, etc. For example, motivated by the central role of communication in WSNs and the success of protocol reference frameworks like ISO/OSI [102] and TCP/IP [20], many proposals have focused on the WSN protocol stack [31, 37, 44, 45, 112, 136, 163]. The protocol stack has also been at the core of the industrial standardization efforts like ZigBee [225], WirelessHART [217] and ISA100 [101], as well as the more recent push for adapting the TCP/IP protocol stack to the needs of WSNs lead by the 6LoWPAN [2] and ROLL [97] IETF working groups. These proposals undoubtedly cover important and necessary aspects of the WSN software stack, but they often have limited scope. Substantial progress along the stated goals of promoting reuse and rapid development is only possible by taking a more general view and by spreading the structured development approach over wider parts of the software stack. Although it is highly unlikely that a single architectural framework will be able to cover the complete diversity in applications and hardware platforms, significant gains are still possible by concentrating on the requirements of several typical classes of WSN systems and leveraging the existing commonalities.

1.3 Double-anchored Software Architecture

This dissertation argues that an effective software architecture for WSNs is indeed possible when right balance between API fixation and composability freedom is struck, at right points in the software stack. We introduce a *Double-Anchored Software Architecture (DASA)* for WSN that enables effective development through traditional methodologies of design and code reuse, using time-tested principles like functional decoupling and complexity hiding. At the same time, it provides mechanisms for controlling the inherent trade-offs between efficiency and reuse so the above benefits can be achieved without paying a prohibitively high price in performance, compared to a customized and vertically integrated solution.

The architecture is specified in the form of a component framework organized around two "anchors" that provide rigidity and offer basis for reuse:

portability anchor that abstracts the hardware while enabling explicit control over the performance-portability trade-offs, and

interoperability anchor that abstracts the communication stack and other node-local services and supports application-specific customization while maintaining application-level interoperability.

Breaking the tight coupling between the software and the underlying hardware through a hardware abstraction layer is a crucial prerequisite for amending the deficiencies of the vertically-integrated development model. By hiding the hardware-dependent code from the rest of the system, hardware abstraction layers facilitate portability and code reuse. Due to these benefits, they have been a standard part of many traditional operating systems [154, 206]. In WSNs, however, hardware abstraction layers often come into direct conflict with the performance and energy-efficiency requirements. In this context the abstraction costs cannot be as easily masked by hardware over-provisioning as in traditional systems, so mechanisms are needed for avoiding some of the abstraction overhead in cases when the need for performance trumps the benefits of the complexity hiding.

The *portability anchor* is our answer to these specific challenges. It codifies the design-constraints that we deem necessary for effective organization of software along the hardware/software boundary. The anchor is structured as a three-level component framework that progressively abstracts the capabilities of the underlying hardware platform. The top level components offer public hardware-independent interfaces for building portable services and applications. At the same time, the middle level components offer public hardware-specific interfaces which provide access to the full capabilities of the underlying hardware. This organization of the hardware abstraction functionality offers several benefits in comparison to a monolithic solution. From one side, it provides a firm base for developing hardware-independent services and applications, allowing significant code reuse across different hardware platforms. From other side, it offers mechanisms for flexible control of the performance penalty for this portability: in situations where the performance loss is too high, the developer can skip the portability abstraction and directly tap to the hardware-specific interface.

While contributing to portability, the portability anchor alone can not significantly improve the productivity of the application development process. Rapid application development needs to be supported by additional complexity hiding through higher-level service Application Programming Interfaces (APIs) that shield the application from the evolution of the underlying service code. In traditional networked systems, this decoupling varies from transparent programming interfaces on top of the communication stack like the Berkeley sockets API [189], to complex interoperability frameworks like CORBA [187]. Both of these extremes seem unsuitable design points for a broad WSN software architecture. From one side, in many WSN application domains, a raw networking APIs is not providing adequate level of abstraction for substantial productivity gains. From the other side, the severe resource constraints make overly complex middleware approaches unattractive.

The *interoperability anchor* in our architecture takes a middle course: it exports a light data-centric abstraction based on the publish/subscribe interaction pattern that is well aligned with the needs of large class of WSN applications. The anchor is organized as a component framework that decouples the exported service from the communication stack and other node-local services. The framework allows easy application-specific optimization of the service through the use of different communication substrates and extension components. This compile-time customization is further complemented by efficient run-time control using metadata attributes.

The two proposed anchors facilitate adaptive enforcement of design-constraints in different zones of the WSN software stack. The mature parts of the stack—where fixation of the interfaces can promote decoupling and reuse—are accompanied by relatively strong design-constraints and composability rules. In contrast—the parts with highest performance impact and prime candidates for application-specific customization—remain highly flexible.

This dissertation contends that a broad software architecture for WSN, anchored at these two levels of the software stack, can effectively promote portability and interoperability while maintaining high sensibility towards abstraction costs. We evaluate these claims in quantitative way using several examples.

1.4 Outline

The rest of the dissertation is organized in six chapters, as follows. In Chapter 2 we provide background information on aspects of the WSN technology and their impact on the organization of software. In the first part, we analyze hardware development trends and review the hardware abstraction support in existing WSN operating systems. In the second part, we overview several programming models for WSN that share similar aims with our interoperability anchor.

In Chapter 3 we introduce the main features of the proposed double-anchored software architecture, focusing on the core organizational principles. Here we provide arguments for the use of the component framework model as effective vehicle for expressing the architectural constraints and we argue about the optimal delineation points in the WSN software stack.

The portability anchor that offers progressive abstraction of the capabilities of the underlying hardware platform is presented in Chapter 4. After discussing the benefits and the specific challenges of abstracting hardware in WSNs, we present the vertical and horizontal decoupling principles of the anchor that provide the needed flexibility. We conclude the chapter by evaluating the effectiveness of the proposed organization using a set of micro-benchmarks and by analyzing the impact from the application of the presented architectural guidelines in the TinyOS 2.x code base, one of the most popular execution environments for WSNs.

The interoperability anchor that exports a customizable data-centric communication service is described in Chapter 5. We discuss the appropriateness of the anchor API as foundation for rapid application development and we present the architectural properties of the anchor that enable application-specific customization of the service by decoupling it from the underlying communication and sensing stacks. In the last part of the chapter we present TinyCOPS, a component framework that implements the anchor and serves as basis for its evaluation.

In Chapter 6 we argue on the need for test-driven development as necessary prerequisite for successful application of the principles of black-box reuse and for exercising the compositional freedom supported by our architecture. We then present the design and implementation of a distributed testing framework that enables efficient testing of the functional and non-functional properties of the data-centric communication service provided by the interoperability anchor.

We conclude the dissertation in Chapter 7 with a summary of the main design features of the proposal and with a reflection on the lessons learned through their application on real-world systems. Finally, we outline several directions of inquiry that can be pursued as follow-up of the presented work.

CHAPTER 2

Background

As a software architecture, the design of DASA is strongly influenced both by the properties of the hardware from below, as well as by the service requirements of WSN applications from above. This chapter provides an overview of the WSN technological landscape and surveys the existing service models that correspond with the two anchors in the DASA architecture.

In the first part of the chapter, we discuss the generic architecture of a WSN node, before proceeding to evaluate in greater detail each of the core hardware elements, focusing on their general role and the design parameters that influence their instantiation on a concrete WSN hardware platform. Using a comprehensive survey of existing WSN platforms, we then analyze the trends in the hardware capabilities over the last decade and discuss their impact on the organization of the software support.

The second part of the chapter overviews existing service abstractions that partially overlap with the intended focus of the two anchors in our DASA proposal: we first analyze the features of the portability abstractions offered by different general-purpose and WSN operating systems, after which we briefly review several programming models and service abstractions for rapid development of WSN applications.

2.1 Hardware Platforms

A successful WSN hardware platform has to achieve a fine balance between providing the necessary functionality, as demanded by the target application, and satisfying the stringent system constraints like size, cost and energy budget. In this section we provide an overview of the hardware architecture of a typical WSN node and a detailed description of five popular WSN node platforms that were used as targets for evaluating the solutions presented in this work. We conclude the section with

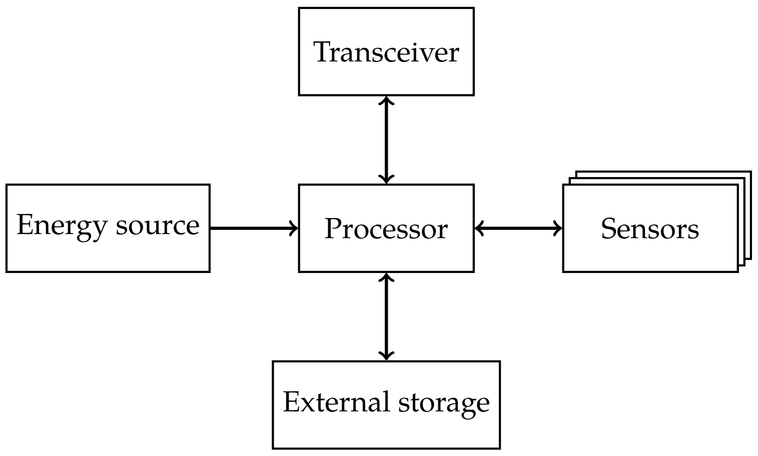


Figure 2.1: Generic hardware architecture of a WSN node

a detailed analysis of the design trends in WSN hardware over the last decade and consider the impact that these trends have on the organization of the system software.

2.1.1 Generic Architecture

Despite a large diversity in target application domains, the hardware architecture of a typical WSN node is fairly regular and reflects the three defining elements of the technology: *sensing*, *computation* and *wireless communication*.

Figure 2.1 shows a schematic representation of this generic architecture. The main elements are the *processor*, the *transceiver*, the *sensors*, the *external storage* and the *energy source*. Although some designs include additional modules like coprocessors, protocol accelerators, dual transceivers, actuators, etc. these five elements are present on the majority of WSN platforms and comprise its technological core.

In the rest of this section we discuss the role that these elements play in the node hardware architecture and the main selection criteria for their instantiation on a specific hardware platform.

Processor

The processor is the cornerstone of the node hardware architecture and is responsible for orchestrating the activities of the remaining elements: it controls the acquisition of data, performs local data processing in preparation for data storage and communication, etc. Due to this central role, the alignment between the application requirements and the processor features is an important prerequisite for a successful platform design.

Having a capable Central Processing Unit (CPU) that can sustain the required processing throughput and limit the length of the active phases can be very beneficial for data-intensive applications needing substantial local computation like signal processing, feature extraction, classification, etc. In these domains, 32-bit embedded CPUs like the Intel PXA27x family [171] are often used, providing high computational power, efficient data transfers and large addressable memory space. These high-end chips also support standard Operating Systems (OSs) like Linux, and offer to the developers a more familiar working environment. Unfortunately, all these benefits come at a significant price in terms of increased energy consumption, component cost and board space, and necessitate careful evaluation of the performance/energy-efficiency trade-offs.

A wide majority of WSN applications, however, are characterized by relatively modest processing needs but require long-term deployments under limited energy budgets. In these scenarios, the nodes spend most of the time in an energy conserving state, interspersed by short periods of high activity to sample new data and to handle communication with other nodes. This mode of operation reduces the importance of the raw processing power of the processing element and brings other features like its start-up time and energy-conservation capabilities to the foreground [134].

In these application domains, there is a tendency of using 8-bit or 16-bit Micro-Controller Units (MCUs). Due to the low-cost and low-power requirements, the MCUs offer limited program and data memory, but dedicate a part of the IC die for additional hardware modules that offload the main CPU and enable higher event handling rates and longer sleeping times. Figure 2.2 shows the functional block diagram of a typical and popular representative of this class of processing elements—the MSP430F161x family from Texas Instruments [194]. The MCU integrates a 16-bit Reduced Instruction Set Computer (RISC) core and a number of additional modules: a watchdog timer, two general-purpose 16-bit wide timers, 12-bit Analog-to-Digital Converter (ADC) and Digital to Analog Converter (DAC), 3-channel Direct Memory Access (DMA), a hardware multiplier and several pin-control and serial-interfacing modules. With 10 KB, this family offers competitive amounts of Random-Access Memory (RAM), a resource that is in high demand in WSNs for message buffering and local preprocessing. The provided flash size (up to 55 KB), is more limited and might be insufficient in applications with large code footprints.

The MSP430F16x family is a good example of the generic set of features necessary for efficient operation in the low-duty cycle regime. It has a flexible clock distribution system, wide timer registers and several DMA channels, maximizing the CPU sleep time. It supports a number of energy conserving states with low RAM retention currents, and has exceptionally fast wake-up time. In addition, the low cut-off voltage helps to fully utilize the energy stored in a battery-based energy source.

Interconnect

The central role played by the processor in the hardware architecture of the WSN node (Figure 2.1) requires an effective and efficient interconnection with the rest of the platform.

The General Purpose Input/Output (GPIO) is one of the most basic and flexible

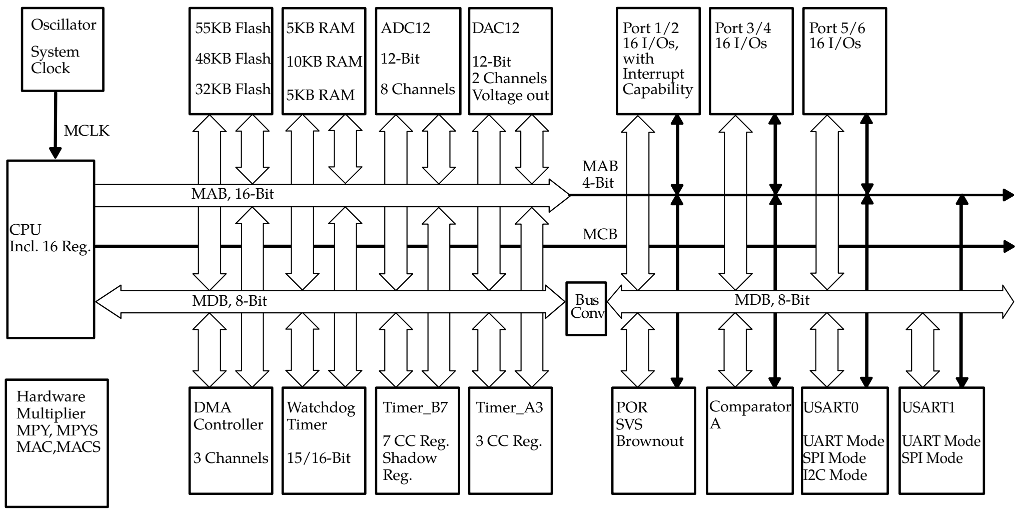


Figure 2.2: Architecture of the Texas Instruments MSP430F161x MCU family.

interfacing options. Each GPIO pin can be addressed either individually or as a member of a larger group of typically 8 or 16 pins, forming a parallel bus. The pins can be used as inputs for reading, or as outputs for sending digital signals to other chips. A typical use is to drive the chip select/chip enable lines of external chips, as addressing support for other more complex interfacing schemes or as power management mechanism. Some GPIO pins can also be configured to generate an interrupt signal to the CPU upon detecting a particular digital signal level or signal transition. This functionality is often used for low-latency signaling, for example, to inform the MCU about the detection of an incoming packet by the transceiver. The main selection criteria for the GPIO interface are the number of GPIO pins, their driving and sinking capability, the leakage currents, the number of pins supporting interrupt generation, and the availability of additional integrated discrete components like pull-up and pull-down resistors.

Although the GPIO interface offers maximal flexibility and minimal latency, to save on pins and connection lines, most of the signaling between the MCU and the external components on the WSN platforms is performed using low-cost serial buses. There are many serial interfaces that can be applied to this end, differing in the number of required signaling lines, the directionality of the communication, the synchronization needs, etc. The most frequently used buses on the WSN platforms are the Universal Asynchronous Receiver Transmitter (UART), the Serial Peripheral Interface (SPI) and the Inter-IC (I^2C).

The UART is a flexible, bidirectional, character oriented, serial interface that does not require explicit clock signal between the sender and the receiver. Instead, the two parties agree on the used data rate upfront and special start and stop bits are inserted in the data stream as framing and to synchronize the receiver before each character reception. In this way, the receiver can sample each individual bit at the right time instance. The lack of explicit clock signal between the sender and the receiver means

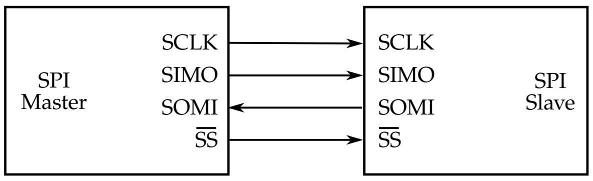


Figure 2.3: The SPI is one of the most popular serial interfaces used on the current WSN platforms. It specifies four logic signals between the master and the slave: a clock line—SCLK; one data line in each direction—SIMO and SOMI; and a slave select line—$\overline{\text{SS}}$

that excessive clock drift between the communicating parties can result in erroneous sampling times, leading to increased error-rates. On WSN platforms, the UART is often used for interfacing with on-board modules that support the Hayes command set (Bluetooth transceivers, serial GPS units, etc.) [109] or for communicating with a PC host through a RS-232 or USB connection.

The SPI is synchronous serial interfacing approach for full-duplex communication between a master and a slave device (Figure 2.3). The protocol offers several different synchronization modes, and imposes no limits on the message size and its content. The SPI can support relatively high data rates, making it well suited for connecting the MCU with high-speed, data-intensive external chips like external ADCs, the transceiver or the external storage.

One of the disadvantages of SPI is the lack of built-in addressing support for configurations involving multiple slave devices. At the cost of a more complex protocol and lower transfer rates, the I^{2}C enables a single master device to communicate with as many as 128 slave devices, using only two signaling lines. On the WSN platforms, the I^{2}C interface is predominantly used for interfacing the MCU with various on-board sensors like accelerometers, light sensors, temperature and humidity sensors, etc.

Due to the wide application of these buses, many MCUs provide built-in modules that implement the necessary signaling and offload the CPU. Very often, these modules are implemented as multifunctional units that can run different serial bus protocols over the same set of pins. This sharing reduces hardware costs, but complicates the support in multi-client scenarios because the software now has to account for the contention and reconfiguration of the hardware module among the different clients.

Important selection criteria for the serial interface modules are the flexibility of the data rate/clock generation system, the control over the frame formats, the level of internal buffering and the interrupt generation capability. On some platforms, thanks to the relative simplicity of the serial protocols, these interfaces are implemented in software, using bit-banging on top of GPIO, at the cost of larger CPU load and increased power consumption.

Sensors

The sensing elements form the interface between the WSN node and the physical reality. The fundamental purpose of the whole WSN node is to act as a carrier for the sensing elements and conduit for the information that they extract from the environment. The type of sensors used on the WSN platforms directly depends on the target application and the physical phenomena that need to be monitored.

The advances in manufacturing, especially the successful repurposing of the semiconductor production process for the creation of Microelectromechanical systems (MEMS), have led to a proliferation of affordable sensing elements which are a perfect match for the specific requirements of the WSN technology. The low cost, however, frequently implies lower fidelity, which has to be mitigated on system level by exploiting the redundancy in the spatial and temporal domain [214].

Sensors export either digital or analog interfaces. In the digital case, their output can be directly read via GPIO, or more complex interfacing can be achieved using SPI or other fast interconnects. In the analog case, the interfacing is achieved through an ADC. The way the sensors are physically connected to the platform also plays an important role in the node design. The sensors can be either co-located with the MCU on the motherboard or placed on a separate daughterboard. Having a standardized electromechanical interface, combined with sensor self-description capability, can lead to a more flexible and reusable platform designs [76, 95].

Many MCU provide integrated ADCs that tend to have low resolution and support modest data sampling rates. Despite these limitations, the integrated ADCs are sufficient for the majority of WSN applications. In more demanding scenarios, requiring higher sampling rates and higher fidelity, external ADC can be used. These dedicated modules are interfaced with the MCU via GPIO-based parallel buses or using fast SPI. Important selection criteria for the ADC are the resolution, the implementation principle, the number of channels, the flexibility in the conversion references, the presence of internal reference voltage sources, etc. The interaction of the ADC with the DMA is also of interest in applications needing high-speed, low-jitter sampling, while keeping the active involvement of the CPU at minimum to conserve power.

The large diversity in WSN applications results in even larger diversity of used sensing elements. According to the survey results published in [57], most popular are environmental sensors for measuring temperature, humidity and pressure, followed by optical sensors and sensors for measuring velocity, acceleration, position and displacement, as well as voltage sensors and current gouges. The majority of these sensors have low accuracy (12 bits on the average) and require modest sampling rates (up to 1 kbps). The respondents of the survey have listed: sensor dependability, longevity, cost, ease of diagnostics, size, operating range and energy consumption—as the most important selection criteria for the sensing elements.

In some applications, the WSN node is tasked not only with sensing the environment, but also with actively influencing it through actuator devices. For digital actuators the interfacing with the MCU can be realized over GPIO or serial buses, while in the analog case, over MCU-internal or external DACs. Main selection criteria for the DAC are the number of supported channels, their resolution and maximal driving current.

Transceiver

The wireless communication is a defining characteristic of the WSN technology. The radio, infrared, and visible light portions of the electromagnetic spectrum can all be used for this purpose by modulating the amplitude, frequency, or phase of the waves.

Communication using Infrared Radiation (IR) is license-free and robust to interference from electrical devices. It is relatively directional, which makes it more resistant to eavesdropping, but also unsuitable for many WSN scenarios where clear line-of-sight is not available. Due to these constraints, only few WSN platforms have used IR as the main communication medium [129]. The visible light part of the spectrum has seen a similarly limited use [208]. The Radio Frequency (RF) is clearly the most suitable transmission medium for the majority of application scenarios [166]. Although they have differentiated communication needs, the majority of WSN platforms use narrowband and wideband radios in the Industrial, Scientific and Medical (ISM) part of the spectrum, benefiting from its license-free nature and the freedom of implementation that it offers [47].

Narrowband radios, with simple modulation schemes like Amplitude-Shift Keying (ASK) or Frequency-Shift Keying (FSK), were the preferred choice on the early WSN platforms [89]. These chips typically offer low level of abstraction, so that the sending and receiving of each bit must be explicitly controlled by the MCU. This provides great flexibility and enables innovation at the Physical Layer (PHY) and Medium Access Control (MAC) layers of the protocol stack. However, it also puts significant load on the CPU which has to take care of low-level aspects like proper bit sampling, encoding/decoding, packet framing, etc. One approach for mitigating this overhead is to use hardware accelerators, either as dedicated custom modules, or by inventive re-purposing of some existing MCU-integrated module. For example, the original mica [91] and the eyesIFX [81] platforms use the SPI and UART modules on the MCU respectively, to offload this processing from the CPU.

One deficiency of radiating the RF energy in a narrow frequency band is the increased sensitivity to interference. By contrast, *wideband radios* can spread their energy over wide frequency bands, using spread-spectrum approaches like Frequency-Hopping Spread Spectrum (FHSS) and Direct-Sequence Spread Spectrum (DSSS), giving them greater robustness to narrowband interference.

The majority of wideband radios that are used on the WSN platforms today are compliant with the Institute of Electrical and Electronics Engineers (IEEE) 802.15.4 standard which defines PHY interfaces based on DSSS and Offset Quadrature Phase-Shift Keying (O-QPSK) modulation. The introduction of the IEEE 802.15.4 standard in 2003 had transformative effect on the selection process for the radio transceiver. Prior to this standardization effort, the WSN platforms used a variety of narrowband and wideband transceivers, making interoperability between different platforms using different transceiver chips extremely hard. The IEEE 802.15.4 standard satisfies the requirements of a broad class of low-power, low-data-rate WSN applications, and has become the preferred solution for the wireless interface on majority of WSN platforms, despite the implied loss of fine-grained access to the PHY [164].

Today, the use of non-standard radios is mainly limited to applications that have very specific communication requirements. Typical examples are outdoor deployments needing long-range links or indoor deployments in challenging environments that can benefit from the better propagation properties in the lower frequency bands. In applications requiring high aggregate data rates and where ranging between the nodes is important, the IEEE 802.15.4a standard finds increased application, thanks to the two PHY layers based on Direct-Sequence Ultra-Wide Band (DSUWB) and Chirp Spread Spectrum (CSS).

The popularity of the IEEE 802.15.4 standard has led to a proliferation of compliant radio transceivers, making the selection of the most appropriate chip for a given platform a daunting task. The designer has to carefully consider a mix of factors that determine the suitability of a given chip. For example, the majority of the chips on the market today offer significantly better receiver sensitivity than the -85 dB borderline defined by the standard [30]. Combined with variability in the maximal transmit power, this provides a range of link budgets that one can select from.

On a typical WSN platform, the radio chip is one of the largest energy consumers. Since the consumption is dominated by the internal radio electronics, these radios typically consume similar amounts of energy in transmit and receive mode. The traditional approach for reducing the energy spent while waiting for reception—the so called “idle listening”—is to duty cycle the radio, keeping it off most of the time and turning it on only to check the channel for an indication of an interested sender [162]. In this regime, the most important selection factors for the radio chip are the wake-up time and the mode switching times.

The signaling and the interconnect are also notable selection factors. The majority of IEEE 802.15.4 compliant radios use the SPI interface for exchanging commands and data with the MCU. In addition, many chips export important internal events on dedicated GPIO pins, like the detection of a Start Frame Delimiter (SFD), the Receive Signal Strength Indicator (RSSI) and the outcome from the Clear Channel Assessment (CCA), thus compensating for some of the lost low-level access.

External Storage

The external storage finds many versatile uses on the WSN platforms. It is an essential architectural element in those applications domains where data sampling occurs at high rates, all data needs to be preserved, but there is an acceptable delay in delivering this data to an external entity. For example, in many health-care applications the selected vital signs like an Electrocardiogram (ECG) can be sampled with high-frequency for a long period, only to be transmitted in bulk at the end of the recording session. Similar use can be found in environmental monitoring and structural health monitoring applications where the nodes proactively transmit only small data summaries over the radio and send the real data-set, that is buffered in the external storage, only on explicit demand [212].

The external storage also plays important role in system administration tasks, as long-term memory for storing configuration information and event logs, as a safe harbor for a "golden" program flash images or as temporary assembly space for Over-The-Air Reprogramming (OTA).

The WSN platforms use external storage based on a number of technologies: NAND and NOR flash, Ferroelectric RAM, etc. Due to the relatively high data rates, the external storage chips usually interface with the MCU via an SPI bus. Important selection criteria for the external storage are the implementation technology, capacity, size of the erasure units, data transfer rate, read and write latency, power consumption in active and sleeping state and wake-up latency.

Energy Source

In some application domains, like smart metering, the WSN nodes have the luxury of an almost endless supply of energy, since they can draw power off the mains power grid [106]. In the majority of applications, however, the nodes have to operate on battery power. This maximizes the deployment flexibility, but at the same time results in a limited energy budget. Frequent replenishing of this budget, by replacing or recharging the batteries, is impractical for many WSN application scenarios due to the large deployment scales and the relative inaccessibility of the nodes. Thus, the energy source has to be dimensioned so that it can cover the energy requirements of the node for the intended lifetime of the application. Conversely, for a limited energy budget, the energy-efficiency of the system determines the maximal obtainable application lifetime.

The battery-based energy source is often the most bulky element on the node, typically determining its total size. This makes the energy density of the battery, a factor of its cell chemistry, one of the most important selection criteria for the energy source.

Given the fact that the load curves in many WSN applications are characterized by high dynamic ranges, the pulsed discharge behavior of the battery also has significant impact on the system design. From one side, some cell chemistries exhibit a recovery effect under pulsed loads, which can be exploited to extend the lifetime of the system [28]. On the other side, many cell chemistries don't support large peak currents very well: a large internal resistance leads to a significant drop in the terminus voltage with serious negative effects for the load electronics.

The cell chemistry also impacts the long-term stability of the terminus voltage as the energy depletes. A flat discharge curve is preferred in many designs, since the load can extract most of the stored energy at near nominal voltage levels. At the same time, such discharge curves complicate the assessment of remaining energy in the source. In chemistries where the terminus voltage drops noticeably as the energy is depleted, determining the remaining energy in the source is possible with a simple voltage sensor, something that most MCU provide on-chip. To guarantee a stable operating voltage regardless of the discharge behavior of the battery, many node designs use voltage regulators, but these introduce significant inefficiencies. Some MCUs, like the presented MSP430F161x, have on-chip brownout protection, that detects the low-voltage condition and can put the system back into a consistent state, by triggering a system reset.

In some application domains, enough power for the operation of the node can be obtained by harvesting the available energy from the environment [161]. Solar energy, mechanical vibration, thermal and pressure differentials, etc. are only few of the energy sources that have been proposed as viable options. In many home automation scenarios, for example, the energy for operating the nodes controlling the lighting can be harvested from the mechanical action of the user as he flips the switch [182]. Since the average power from the harvesting sources is low, to handle the peak current draws, a temporary energy storage is often needed in the form of a rechargeable battery or a large capacitor [104].

2.1.2 Typical Representatives

In this section we provide a more detailed overview of the features of five mature, commercially available and popular platforms: *mica2*, *micaz*, *telosb*, *eyesIFXv2.1* and *intelmote2*. [1] They offer a representative sample of the hardware design space (Section 2.2) and have been used for evaluation of aspects of the DASA design, as presented in the subsequent chapters.

mica2

After experimenting with highly integrated Application-Specific Integrated Circuit (ASIC) solutions like *Spec* [90], in the context of the SmartDust project [208] the researchers at the University of California, Berkeley, in the late 1990s, turned their focus to building more generic WSN prototyping platforms using Commercial Off The Shelf (COTS) components. In this effort to approximate the envisioned hardware capabilities of a WSN node with affordable components, the Berkeley researchers have produced a long line of hardware designs, called *motes* that became a de-facto template of what a WSN node is.

The *mica2* [89] is one of the first platforms from the Berkeley motes that achieved widespread use after its commercialization through Crossbow Technology. Following in the footsteps of the *WeC*, *Rene* and the original *Mica* design [91], the mica2 uses a processing element from the Atmel ATmega family, a simple 8-bit RISC architecture with Harvard memory organization. With 128 KB flash, the ATmega128L provides

[1] There is wide variation in the capitalization and spacing of the platform names in the literature, here we use the same format as in the TinyOS code-base.

mica2		
UC Berkeley and Crossbow Technology		
Processor	*Atmel ATmega128L*	
	Clock frequency	7.4 MHz
	Flash	128 KB
	RAM	4 KB
	Current consumption	8 mA
	Minimum operating voltage	2.7 V
Transceiver	*Chipcon CC1000*	
	Modulation	FSK
	Frequency band	315/433 MHz; 868/916 MHz ISM
	Data rate	38.4 kbps
	Current consumption	27 mA (TX); 10 mA (RX)
	Minimum operating voltage	2.1 V
External storage	*Atmel AT45DB041B*	
	Capacity	512 KB
	Current consumption	10 mA (R); 35 mA (W/E)
	Minimum operating voltage	2.7 V
Energy source	*2x AA batteries*	

Table 2.1: Summary of the mica2 platform features.

ample space for the program image, sufficient even for more complex stacks and application code. The RAM size, with only 4 KB is much more constrained and can be a limiting factor in solutions requiring significant amounts of message buffering or local data processing.

In contrast to the previous Mica designs featuring a bit-oriented TR1000 radio, interfaced through hardware accelerators (Section 2.1.1), the mica2 uses a Chipcon CC1000, a more reliable byte-oriented radio interfaced via a SPI bus. The CC1000 is a FSK transceiver chip with no internal buffering and supporting relatively low data rates. The radio is attached to a simple $\lambda/4$ whip acting as monopole antenna.

Compared to the original Mica, the mica2 also brought increased external data storage space in the form of an Atmel AT45DB041B serial flash chip, providing additional room for data logging and OTA. The platform does not have on-board sensors, but offers a 51-pin extension connector that can be used to attach external sensor-boards.

micaz

The *micaz* is the latest design from the Mica mote line. It maintains the same processing element, external storage and extension connector as the mica2 predecessor.

The main innovation is the inclusion of a wide-band IEEE 802.15.4 compliant radio (Section 2.1.1), in the form of a Chipcon CC2420 chip [30]. The new radio raises

micaz **UC Berkeley and Crossbow Technology**		
Processor	*Atmel ATmega128L*	
	Clock frequency	7.4 MHz
	Flash	128 KB
	RAM	4 KB
	Current consumption	8 mA
	Minimum operating voltage	2.7 V
Transceiver	*Chipcon CC2420*	
	Modulation	O-QPSK
	Frequency band	2.4 GHz ISM
	Data rate	250 kbps
	Energy consumption	17.4 mA (TX); 19.7 mA (RX)
	Minimum operating voltage	2.1 V
External storage	*Atmel AT45DB041B*	
	Capacity	512 KB
	Current consumption	10 mA (R); 35 mA (W/E)
	Minimum operating voltage	2.7 V
Energy source	*2x AA batteries*	

Table 2.2: Summary of the micaz platform features.

the level of abstraction, providing a convenient packet-based interface at the cost of the direct access to the PHY of the previous Mica designs. It offers many internal hardware accelerators that offload the work from the MCU, while giving direct access to time sensitive information like the SFD and other PHY parameters like the RSSI and the Link Quality Indicator (LQI).

telosb

We use the name *telosb* as generic name for all sub-variants of this platform. After the original development at UC Berkeley, the telosb developers commercialized an improved version under the name *Tmote Sky*. The old Berkeley design is still available from Crossbow Technologies as *TelosB*.

The design of the Telos mote line [164] focuses on integration of many on-board components while maintaining low-power operation. This is achieved through higher level of isolation between the components, enabling individual power supply control and increased reliability.

The Telos motes and the WSN nodes developed in the EYES project [51] were one of the earliest commercially available platforms using the Texas Instruments MSP430 MCU family. As discussed in Section 2.1.1, this architecture is optimized for fast wake-ups and has very flexible low power modes, making it well matched to the specific WSN platform needs. In contrast to the first designs in these lines that used

telosb **UC Berkeley, Crossbow Technology, Moteiv Corporation**		
Processor	*Texas Instruments MSP430F1611*	
	Clock frequency	4 MHz
	Flash	48 KB
	RAM	10 KB
	Energy consumption	1.8 mA
	Minimum operating voltage	1.8 V
Transceiver	*Chipcon CC2420*	
	Modulation	O-QPSK
	Frequency band	2.4 GHz ISM
	Data rate	250 kbps
	Energy consumption	17.4 mA (TX); 19.7 mA (RX)
	Minimum operating voltage	2.1 V
Onboard sensors	*Light, Temperature, Humidity*	
External storage	*ST Microelectronics STM25P80*	
	Capacity	1 MB
	Current consumption	4 mA (R); 15 mA (W/E)
	Minimum operating voltage	2.7 V
Energy source	*2x AA batteries; USB*	

Table 2.3: Summary of the telosb platform features.

the MSP430F149 MCU with only 2 KB of RAM, the telosb features a MSP430F1611 MCU with 10 KB of RAM which provides support for more memory demanding software solutions.

Like the micaz, the telosb uses the Chipcon CC2420 radio, but it has an integrated inverted-F micro-strip antenna. Both the transceiver and the STM25P80 used as external storage, share the same SPI interface to the MCU, requiring suitable arbitration in software.

A significant innovation of the Telos design was the inclusion of an Universal Serial Bus (USB) interface allowing power supply, programming, and data transfer to be performed over the same connector. This simplifies the interfacing with external devices, making it the preferred System Under Test (SUT) platform for many WSN testbeds (Section 6.2.1).

The platform supports several on-board sensors: two light sensors and a humidity and temperature sensor, in addition to the internal sensors for voltage and temperature provided by the MCU. Additional sensors can be interfaced through an expansion connector. For example, the Tmote Invent platform [147] features a sensor daughterboard with light sensors, microphone, speaker and accelerometers that is connected to a Tmote Sky node serving as a motherboard.

eyesIFXv2.1 **Infineon Technologies AG**		
Processor	*Texas Instruments MSP430F1611*	
	Clock frequency	4 MHz
	Flash	48 KB
	RAM	10 KB
	Energy consumption	1.8 mA
	Minimum operating voltage	1.8 V
Transceiver	*Infineon TDA5250*	
	Modulation	ASK, FSK
	Frequency band	868 MHz ISM
	Data rate	19.2 kbps
	Energy consumption	12 mA (TX); 9 mA (RX)
	Minimum operating voltage	2.1 V
Onboard sensors	*Light, Temperature*	
External storage	*Atmel AT45DB041B*	
	Capacity	512 KB
	Current consumption	10 mA (R); 35 mA (W/E)
	Minimum operating voltage	2.7 V
Energy source	*1x CR2477 battery; USB*	

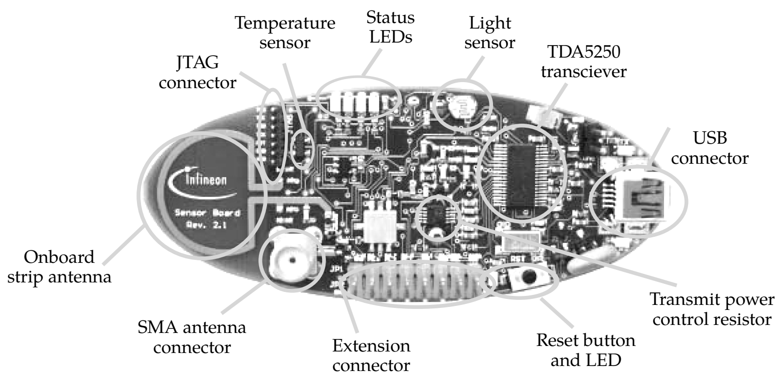
Table 2.4: Summary of the eyesIFXv2.1 platform features.

eyesIFXv2.1

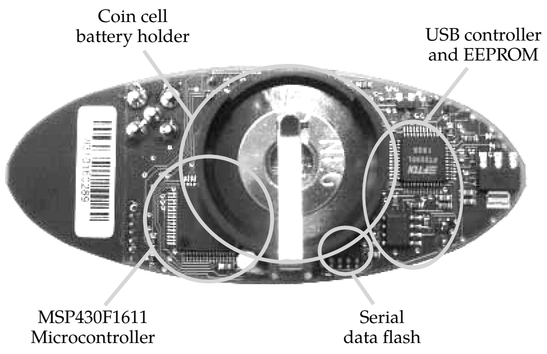
The *Eyes* line of node designs has been developed in the framework of the European research project EYES [51, 84]. Similar to the Telos line, the Eyes nodes use Texas Instruments MSP430 MCUs. The node line developed by Nedap incorporated the same RF Monolithics TR1001 transceiver like the original Mica motes. The *eyesIFX* line, developed by Infineon Technologies AG in cooperation with TU Berlin, use an Infineon TDA5250 transceiver.

The *eyesIFXv2.1* is the latest node design in the eyesIFX line. It was sold by Infineon as part of a development kit with eight sensor nodes and a base station. Our group has been the core developer and maintainer of the software support suite for the kits, based on our MSP430 abstractions included in the TinyOS 1.1.7 release [81].

Like the telosb platform, the eyesIFXv2.1 uses an MSP430F1611 as processing element. It has the same external storage chip like the mica platforms and the telosa. Figure 2.4 shows the characteristic oval shape and the main components of the board. The top side is dominated by the TDA5250 chip, an ASK/FSK transceiver supporting speeds up to 64 kbps. The TDA5250 is a general-purpose narrow-band radio normally used at low data rates in automotive applications like remote key-less entry. The radio operates in the 868-MHz ISM band which offers good indoor propagation char-



(a) The top side is dominated by the Infineon TDA5250 transceiver and the expansion connector.



(b) The TI MSP430F1611 microcontroller and the FTDA USB controller are the largest chips on the bottom side, partially obscured by the coin cell battery holder.

Figure 2.4: Top and bottom view of the eyesIFXv2.1 board.

acteristics thanks to the lower frequency and the reduced interference in comparison to the more crowded 2.4 GHz ISM band.

The radio provides SPI and I^{2}C control interfaces, while the data interface remains bit-oriented like in the original Mica designs. For reduced latency, several functions of the transceiver like the modulation type, the switching between the Receive (RX) and Transmit (TX) mode, etc., can be directly controlled using GPIO. The radio also directly exports several PHY parameters allowing flexible experimentation with PHY and MAC protocols.

The interfacing of the data line of the radio is the most interesting characteristic of the platform. To reduce the interrupt load generated by separately handling every data bit, the UART module from the MCU is used as hardware accelerator to build a byte interface on top of the bit stream. In this way, the eyesIFX is similar to the original mica mote and its use of the SPI interface for the same purpose. Each radio

intelmote2 **Intel Corporation, Crossbow Technology**		
Processor	*Intel Corporation PXA271*	
	Clock frequency	13–416 MHz
	Flash	32 MB
	RAM	32 MB
	Energy consumption	48 mA at 104 MHz and 0.95 V core voltage
	Minimum operating voltage	0.85 V
Transceiver	*Chipcon CC2420*	
	Modulation	O-QPSK
	Frequency band	2.4 GHz ISM
	Data rate	250 kbps
	Energy consumption	17.4 mA (TX); 19.7 mA (RX)
	Minimum operating voltage	2.1 V
Energy source	*3x AAA batteries; USB*	

Table 2.5: Summary of the intelmote2 platform features.

packet starts with a training preamble followed by a SFD that is constructed so that it re-synchronizes the receiving UART for the reception of the data bytes, relying on the fact that the sending UART always transmits an 0xFF pattern when there is no data. Each data byte is encapsulated in one start and one stop bit which also help to satisfy the DC-balancing requirements of the circuit.

In addition to the internal MCU sensors, the eyesIFXv2.1 provides on-board light and temperature sensor. An expansion connector can be used for interfacing with external boards. In contrast to the USB-A connector used in the Telos designs, it uses a mini-USB, female connector, to reduce the size impact and to improve the mechanical stability of the board.

Another specific of the eyesIFX platform is the use of a lithium-ion coin cell battery as the main energy source, similar to the mica2dot platform. The CR2477 requires small mounting space and provides about 1000 mAh of energy. Unfortunately, it also has a rather low peak current limit (Section 2.1.1) which can have negative impact on the system lifetime.

intelmote2

In contrast to the previous four platforms, the *intelmote2* [3] design targets high-performance sensing applications, like industrial monitoring, with large local data processing and communication needs. It is also frequently used as a gateway platform for connecting WSNs with external networks [152]. The platform is commercially available from Crossbow under the name *Imote2*.

The intelmote2 is a successor of the original *Intel Mote* or *IMote* design [151], developed by Intel Research Berkeley Lab in cooperation with UC Berkeley. It is a highly integrated platform, offering significant raw computing power in the form of a PXA271 CPU featuring an Intel Xscale 32-bit core with 32 MB of flash and 32 MB of RAM. The ample resources enable the use of traditional operating systems like Linux or the Microsoft .NET framework, providing a more traditional system environment for the software developers.

The PXA271 also integrates a Digital Signal Processing (DSP) coprocessor and supports a rich set of Input/Output (I/O) interfaces. In addition, it has a voltage scaling feature allowing dynamic control of the core supply voltage, so that applications can trade off performance and power consumption which is one to two orders of magnitude higher than on Telos and Eyes.

In contrast to the original imote, which had a Zeevo TC2001P Bluetooth transceiver for bandwidth hungry applications, the intelmote2 uses the same Chipcon CC2420 radio as the micaz and telosb. The core platform is designed as pure computational and communication board, without any on-board sensors. It has stackable connectors on the bottom and the top side, allowing easy integration of additional sensor and power boards. Like the eyesIFXv2.1, it uses a mini-USB connector for interfacing with a host computer.

2.2 Hardware Design Trends

Despite the relative simplicity of the general architecture, the various selection criteria for the individual hardware components elaborated in Section 2.1.1, result in a very wide design space. To detect the main hardware trends and analyze their impact on the software architecture we surveyed over ninety WSN platforms introduced over the last decade in both academia and industry. The complete list of surveyed platforms including the characteristics of their core components is presented in full in Appendix A.

Figure 2.5 depicts the release year structure of the collected platform sample. It illustrates the rapid evolution in the WSN platforms in the last decade, with tens of new platforms released each year. The fact that the distribution peaks in year 2005 is not a result of a recent reduction in the interest in the WSN technology, but a consequence of the our sample collection process. Namely, we mostly relied on primary academic papers and secondary sources like survey papers and books (listed in Appendix A), producing a sample that is biased towards more mature platforms. Some of the most recently released commercial platforms are not sufficiently covered in the used literature and are consequently less represented in the sample.

2.2.1 Level of Reuse and Integration

Given the specific focus of WSN technology, it might be expected that highly customized, ASIC solutions would dominate the WSN hardware landscape. The associated high design and production overheads, however, make them less commercially attractive and our survey shows only a few such examples. On the contrary, the

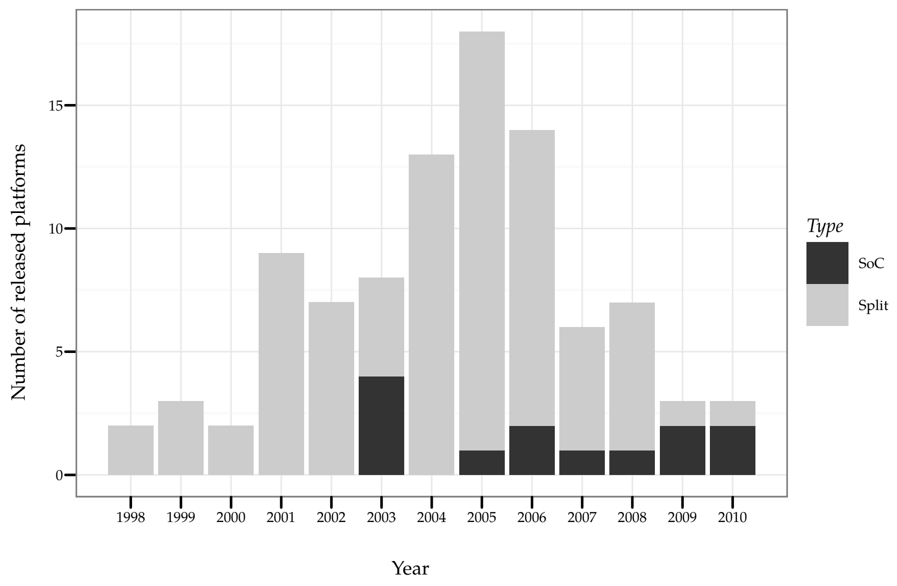


Figure 2.5: Overview of the release year and the level of integration of the platforms covered by our hardware survey.

use of COTS components and the resulting economies of scale have been the most common approach for reducing the unit prices of the WSN nodes. Consequently, the majority of the surveyed platforms are build from readily available components.

We analyze this trend by investigating the "chip reuse" histograms that depict the number of platforms on which a given hardware chip has been used. The results show that for the processing element and the transceiver, these histograms resemble a power-law curve: a small number of popular chips are used on many platforms, at the same time, there is a heavy-tail of chips that are only represented on a single platform. Figure 2.6 highlights the ten most popular processor/transceiver combinations in the survey sample. In comparison to these core elements, the diversity in the peripheral chips and sensing elements is much larger.

With the increased use of the CMOS technology in the transceiver production process, new opportunities were opened for further miniaturization and integration, in the form of System-on-Chip (SoC) solutions that bolt the transceiver together with a full MCU core. Despite the bias of the survey sample, Figure 2.5 highlights the increased interest in platforms based on such highly-integrated components. Typical examples of this trend are the plethora of solutions that have emerged lately on the market which combine an IEEE 802.15.4-compatible transceiver with a simple 8-bit MCU core, like the Ember EE250 or the Texas Instruments CC2430. The popularity of the Intel MCS 8051 processing core in these offerings illustrates the increasing importance of open designs and low royalties as selection criteria in this price sensitive domain.

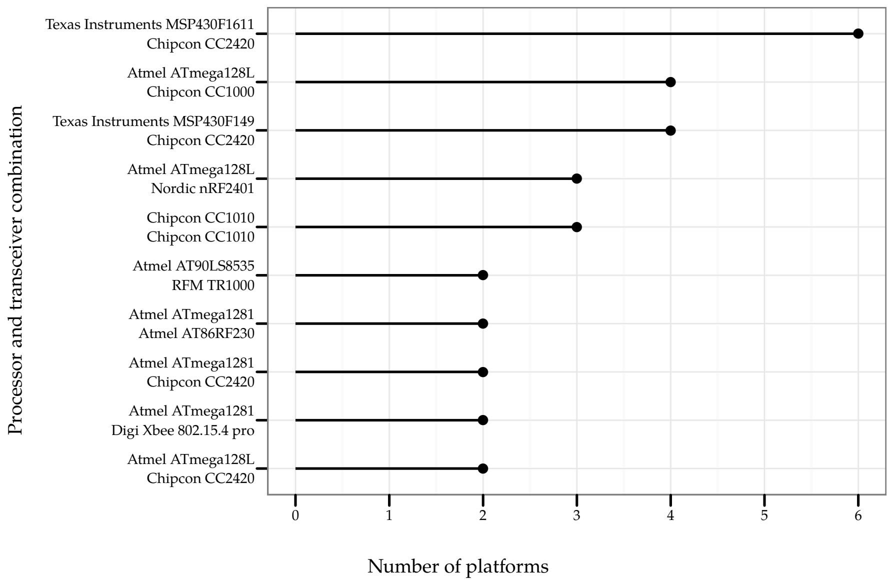


Figure 2.6: Level of reuse of COTS chips in the surveyed platform sample. Only the ten most popular processor/transceiver chip combinations are shown.

2.2.2 Feature Trends

The trending analysis of the hardware parameters of the surveyed platforms shows that, over the last decade, there has been a relatively slow growth in the capabilities of the nodes. Instead, most of the Moor's Law gains were translated into reduction of the size, cost and power-consumption parameters.

In the remainder of the section, we illustrate these trends by analyzing the evolution of the available resources on the surveyed platforms, focusing on the processing element and the transceiver, as core elements of the node hardware architecture. To extract the short-term trending information from the collected data set, we augmented the raw time/parameter scatter-plots with Locally Weighted Scatterplot Smoothing (LOESS) curves [34] including their 95% confidence interval regions. For visualization of the long-term trending we used simple linear model fitting.

Processing Element

Figure 2.7 shows a breakdown of the number of surveyed platform releases each year depending on the "bitness" of the CPU. The bit-width of the architecture can be taken as indicator for complexity of the processing element, and by that it can shed some insight into the computational needs of the target application.

In the collected platform sample, the majority of platforms were equipped with 8-bit and 16-bit CPUs, confirming the relatively moderate data-processing demands in many WSN application domains. The data shows increased use of 16-bit CPUs (mainly represented by the Texas Instruments MSP430 MCU) in the second part of the decade.

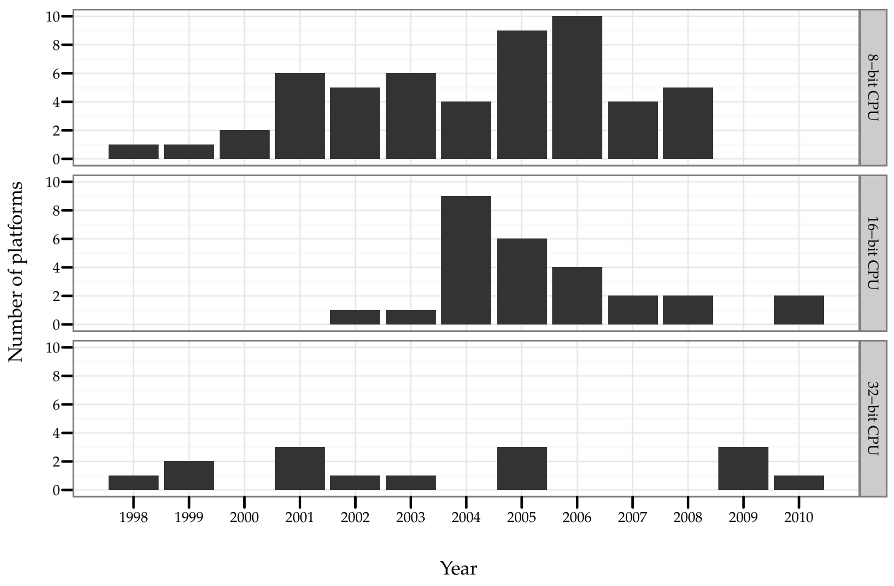


Figure 2.7: Breakdown of platform releases per year depending on the bit-width of the CPU.

Although in minority, platforms leveraging more capable 32-bit CPUs have been produced throughout the surveyed time window, indicating the need for higher computational power in some specific high-end WSN application scenarios.

In addition to the bit-width, for each of the surveyed platforms, we also collected information about the maximal clock rate, minimal core voltage, as well as the amount of available program and data memory. An initial analysis of the data set has indicated strong clustering of these parameters based on the bit-width of the architecture, making a joint trending analysis over all platforms misleading, as the data exhibits more than an order of magnitude variability across the different bitness groups. To highlight this internal structure, in the following analysis we have resorted to faceting based on the bitness.

Figure 2.8 shows the results for the trending analysis of the maximal clock rate. For the surveyed platforms having an 8-bit or 16-bit processing elements, we can see a slow trend toward higher clock rates, but the processing power on these platforms remains relatively constrained and does not exceed low tens of MHz. For the 32-bit platforms, the trend seems to even go towards lower CPU frequencies, but the small number of such platforms in the sample prevents making strong conclusions.

Figure 2.9 shows the trending in the minimum allowable supply voltage of the processing core. In addition to the clock rate, the supply voltage of the core is the main determinant of the power efficiency of the processing element. The results show clear improvement over the last decade for the 8-bit platforms, bringing the minimum supply voltages for the CPU down in the sub 2 V region, on par with the capabilities of their 16-bit and the 32-bit counterparts.

The developments in the available program and data memory are illustrated

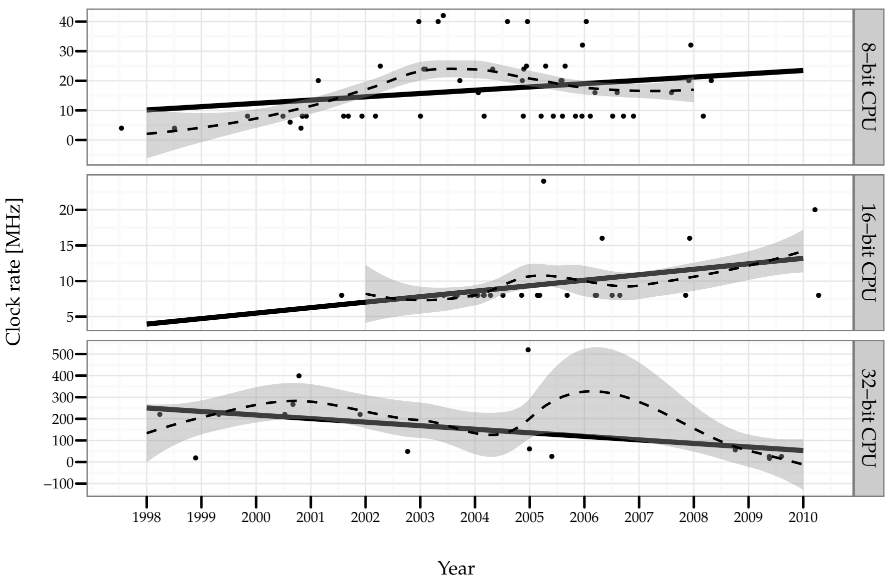


Figure 2.8: Trends in the maximal clock rate of the CPUs, grouped by the bit-width of the architecture.

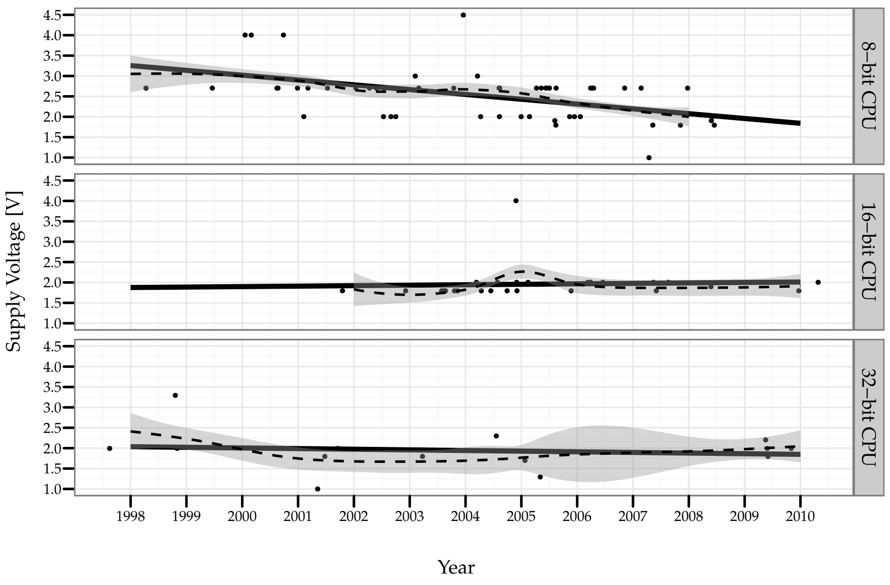


Figure 2.9: Trends in the core voltage of the processing elements, grouped by the bit-width of the architecture.

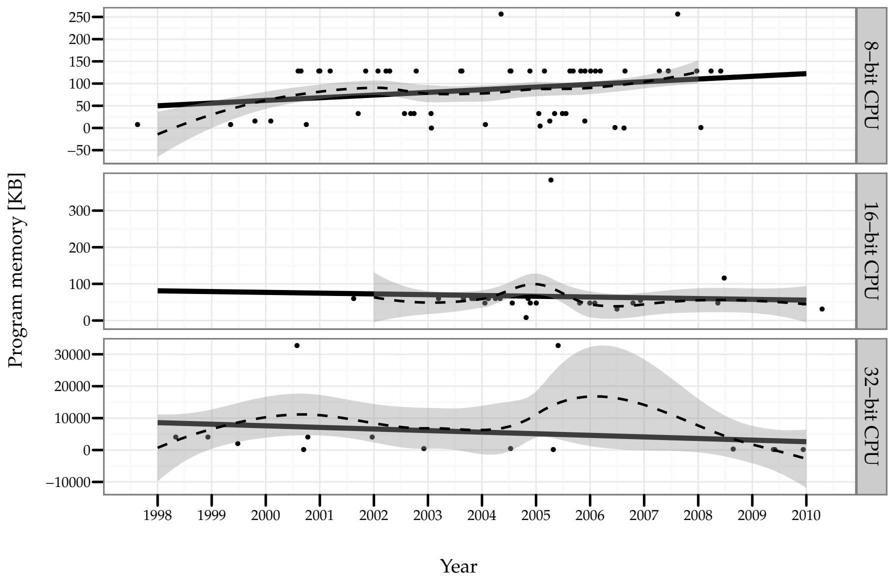


Figure 2.10: Trends in the available program memory of the processing elements, grouped by the bit-width of the architecture.

on Figure 2.10 and Figure 2.11, respectively. The trending analysis confirms that memory remains one of the most constrained resources on the 8-bit and 16-bit WSN platforms, which comprise the majority in our sample. The program memory shows a very weak growth tendency, but still remains significantly constrained with typical values in the range between 50–150 KB. The data memory is even more scarce: over the last decade it managed to grow up only into the range of about 10 KB, exhibiting similar growth rate as the program memory. Even for the 32-bit platforms, which have one to two orders of magnitude larger memory resources, the trend does not show substantial increase over the surveyed period.

Transceiver

Turning our attention to the communication subsystem, Figure 2.12 shows a breakdown of the number of platform releases per year in the sampled set, depending on the communication standard supported by the used transceiver chip.

The distribution shows a strong interest in non-standard compliant transceivers, throughout the surveyed period. The rapid rise in popularity of the IEEE 802.15.4 standard in the second part of the decade is well reflected in the sample set, and the majority of platform releases after year 2004 are using transceivers supporting this standard. Another wireless standard with substantial use is Bluetooth. Despite loosing ground to the 802.15.4-based solutions, it is still often used as secondary communication technology, on platforms supporting two transceivers. In contrast, the number of platforms using WLAN and DECT compliant transceivers is almost negligible in our sample.

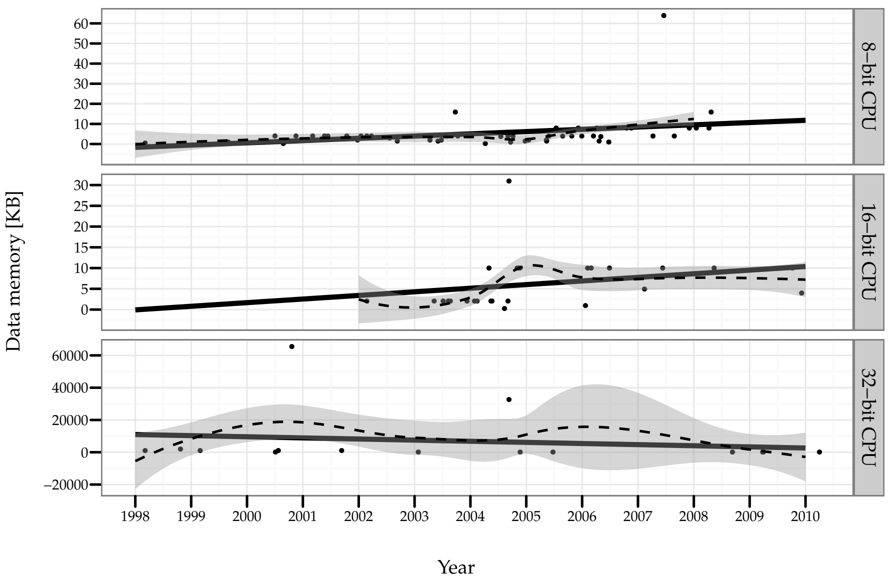


Figure 2.11: Trends in the available data memory of the processing elements, grouped by the bit-width of the architecture.

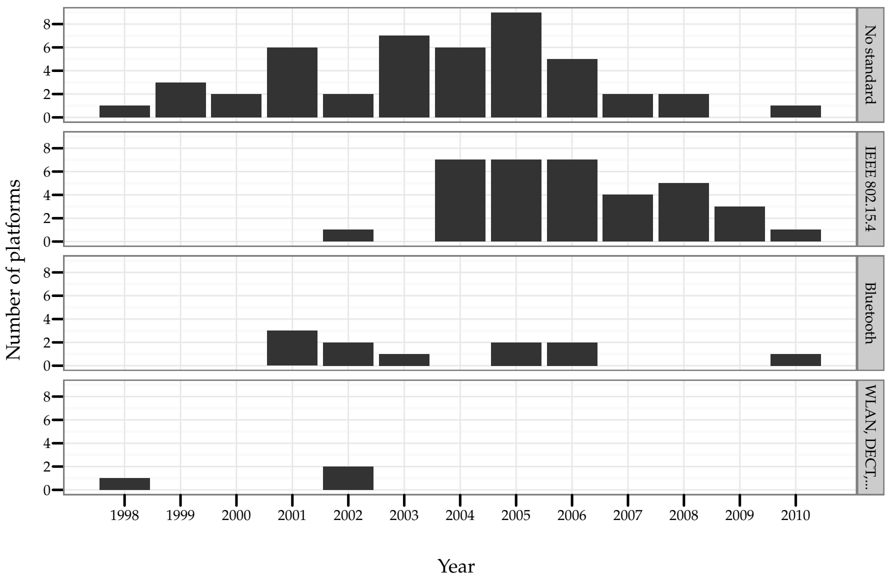


Figure 2.12: Breakdown of platform releases per year depending on the communication standard supported by the transceiver.

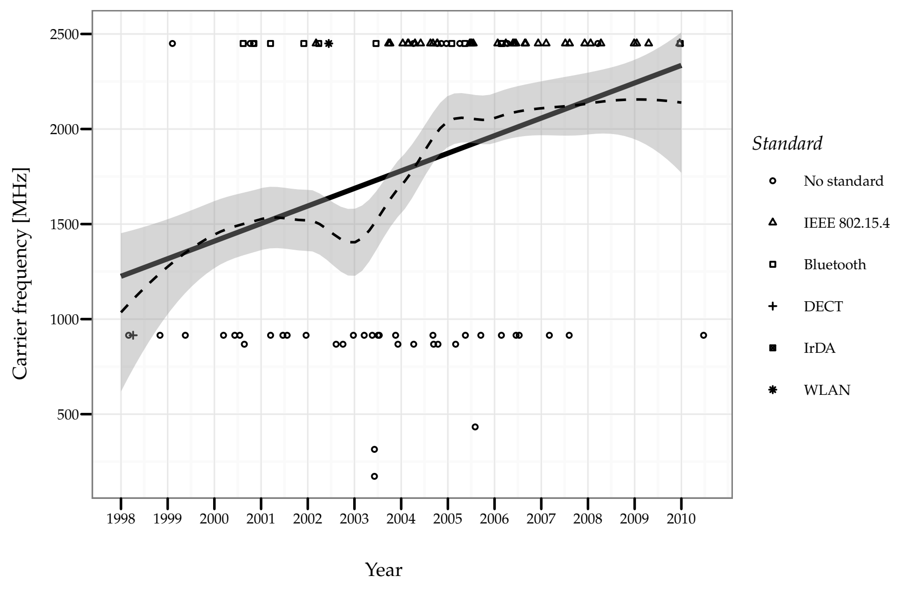


Figure 2.13: Trends in the maximal carrier frequency band supported by the transceiver.

To evaluate the trends in the capabilities of the transceiver chip, in addition to their standard compliance, we have collected information about the frequency band and the maximal data rate and current draw. The trending analysis was performed following the same procedure as for the processing element, using a combination of a LOESS fitting for the small-scale trends and a linear model for the global-trend. Due to the dependence of the transceiver parameters from the supported communication standard, we highlight the standard compliance for each sample with different symbols in the presented raw scatter-plot. In addition, the points in the plot are slightly horizontally jittered to make the overplotting (from the multiple platform releases in each year) more evident.

Figure 2.13 depicts the trends in the operational frequency of the transceiver. For the chips supporting multiple operational bands, we have selected the highest frequency band as representative for the analysis. The results mirror the information in the standards histogram, since the standard constraints the operational frequency. Almost all non-standard compliant transceivers in the sample operate in the 868 MHz and 915 MHz ISM bands. The popularity of the IEEE 802.15.4 transceivers is evident in the jump of the small-scale trend line towards the 2.4 GHz ISM band, after year 2003. The overplotting in the jittered data points is a fitting image of the crowdedness that exists in the band from the different coexisting wireless technologies. The resulting interference problems are one of the main reason for a recent resurgence in the interest for the sub 1 GHz ISM bands that also offer better indoor propagation characteristics.

Figure 2.14 summarizes the developments in the maximal supported data rates. Over the surveyed period, the typical data rate on the WSN platforms has remained relatively low and has not followed the rapid growth in speeds characteristic for

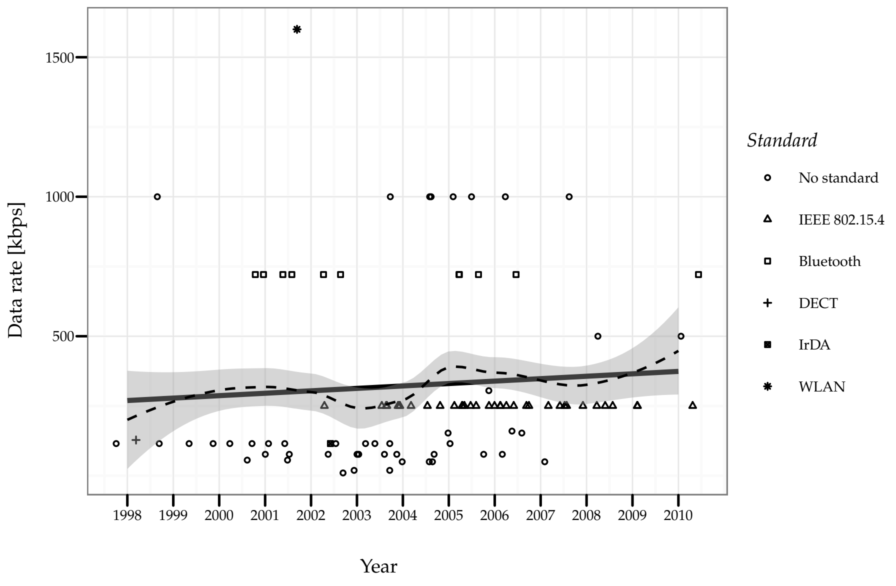


Figure 2.14: Trends in the maximal data rate supported by the transceiver.

other domains like wireless access and home networking, highlighting the different application focus. In the early years of the WSN technology, the majority of the released platforms featured non-standard compliant transceivers with very low data rates. The slow growth trend, introduced by the use of the IEEE 802.15.4 compliant transceivers the with their standard 250 kbps rate, has been recently additionally strengthened by platforms using modern, efficient non-standard compliant transceivers which support higher data rates (500–1000 kbps). With the higher speeds, the new chips can facilitate further shortening of the application's active phases, leading to lower duty cycles and better system lifetimes.

For this, the energy penalty of the increased speed should not undermine the potential gains in the shorter active times. Figure 2.15 shows the trend in the current draw of the transceiver on the surveyed platforms, while sending at maximal TX power. The results confirm that the energy consumption of the chips has remained relatively stable, moving in the range of few tens of milliamperes. Furthermore, the increase in complexity and speed brought by the newer radios have been achieved without prohibitive increase in the energy footprints.

2.2.3 Software Impact

This results from our platform survey indicate that, in foreseeable future, the hardware on the WSN platforms will continue to be optimized for the specific deployment scenarios, resulting in a wide hardware design space. To achieve the desired improvements in the software development process that we motivated in Chapter 1, we need to introduce flexible software support that can adjust to such application-specific

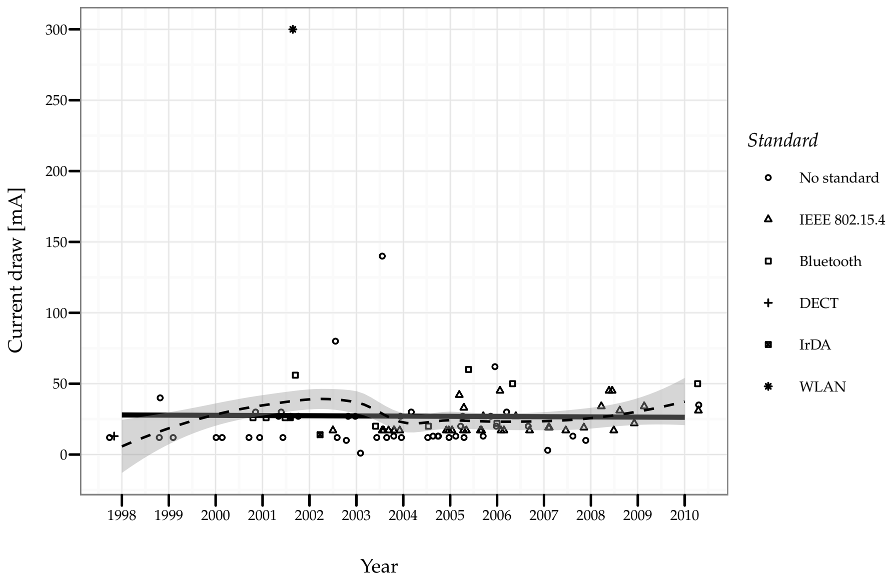


Figure 2.15: Trends in the maximal current consumption of the transceiver.

optimizations while providing a stable base for a more rapid application development. We need a strong base in the software architecture that will help in reducing the dependence from the underlying hardware and will open a way for platform independent development of services and applications, as necessary prerequisite for breaking the tight vertical integration that is typical for the existing solutions. The rapid development in the WSN hardware, illustrated by the results in our survey, highlights the value of such a software base serving the role of a portability abstraction that protects the investment in developed code, by allowing easy reuse and migration to new hardware.

The survey also provides insight in the preferred internal structure of this abstraction layer. The rapid development of the hardware and the moving hardware/ software boundary requires solutions that effectively mask this dynamics from the remaining code in the system. Furthermore, the popularity of the COTS composition approach motivates approaches that enable mirroring of this hardware design sharing in the software process, resulting in reduced porting costs for new platforms with common hardware elements.

The trending analysis confirms that, over the last decade, the processing power of the WSN platforms has grown much slower than what is made possible by Moor's Law. Instead, the gains have been used to optimize other system parameters like size, cost and power consumption. Extrapolating from this trend, we can expect that in the near future a large class of the WSN platforms will remain severely resource limited. Thus, the execution environment will have to provide means for accessing the hardware through lean abstractions that provide complexity hiding, but still allow full control over the hardware when this is needed. The tightly bounded memory resources also motivate rich compile-time customization and adaptation vs. memory expensive run-time introspection and control.

2.3 Portability Architectures

One of the major roles of an execution environment is to create a unified abstract computing environment for the application programmer that is independent from the details of the underlying hardware. For this, the parts of the software that directly interact with the hardware have to be isolated from the parts that have more general applicability and can be reused over different platforms.

This hardware transparency is traditionally achieved with two related concepts: an *Hardware Abstraction Layer*[2] that mostly deals with the architectural differences of the processing units; and a *Device Driver Model* that concentrates on the abstraction and the interfacing of peripheral hardware devices.

Although widely used, the realization of these concepts varies significantly among the different OS and can lead to different trade-offs between efficient resource use and portability. Before introducing our own framework in the form of the portability anchor of DASA (Chapter 4), in this section we briefly review several prominent portability architectures from the general-purpose computing and embedded systems domain, including some existing WSN solutions.

2.3.1 General-purpose and Embedded Operating Systems

Because they have to operate on top of a large number of complex and diverse processing units, the general-purpose and embedded OSs typically have mature hardware abstraction layers. To hide the architectural differences across the different processing units, the OS provides processor-independent mechanisms for handling interrupts, exceptions, memory paging, I/O, Inter-Process Communication (IPC), etc.

The organization of the device drivers, in turn, is closely related with the abstraction of the chip interconnect (Section 2.1.1). For example, the *NetBSD* is claimed to be the most portable of the modern UNIX-like operating systems. It currently runs on top of more than fifty different hardware architectures. This extraordinary portability is due to the specific design of the device driver framework which is machine-independent and based on clean separation between the chip drivers and the bus interfacing code. The access to the bus memory and register areas is implemented in a fully machine-independent way, allowing the same device driver source to be used on different system architectures and bus types [196]. The new *Linux* device driver model [145], reuses a subset of these architectural approaches and structures the hardware abstraction functionality in different elements like *buses*, *classes*, *devices* and *drivers*.

[2]the typical acronym used is *HAL*, but in this work, this acronym is reserved for the *Hardware Adaptation Layer*, part of the vertical decomposition architecture of the portability anchor (Section 4.2.2).

Another important dimension in the organization of the hardware abstraction code is the level of adaptability, i.e. the degree to which the hardware abstractions code can be adapted to the specific needs of the application at hand. The embedded OS *eCOS* is a typical example for a more adaptable organization of the hardware abstraction functionality. eCOS is component-based and uses compile-time reconfiguration to trim the execution environment to the specific requirements of each application.

The hardware abstraction functionality in eCos is organized in two main parts [138]: an "abstraction layer" that provides architecture-independent support for handling interrupts, virtual vectors and exceptions; and a group of "device drivers" that abstract the capabilities of the hardware modules. The drivers are implemented as *monolithic* components and are accessed by the rest of the system via the "I/O Sub-System" that defines a standard interface for communication with the exposed driver "handlers".

In contrast, the device drivers in the *WindowsCE* embedded operating system from Microsoft can be either monolithic or structured in two customized layers [150]. The upper layer is formed by the platform-independent "Model Device Driver (MDD)" that uses the services of the "Platform-Dependent Driver (PDD)" that forms the lower layer. Unfortunately, the application is not allowed to directly access to the lower Platform-Dependent Driver interface and has to communicate with the hardware via single "Device Driver Interface (DDI)".

In Section 4.1 we discuss why the monolithic organization of the driver components like in eCOS, nor the two-layer model of WindowsCS, fully addresses the specific requirements of WSNs, and the need to balance between system efficiency and portability on these severely resource-constrained devices.

2.3.2 WSN Operating Systems

As discussed in Section 2.1.1, most of the WSN platforms use relatively simple MCUs that don't offer hardware-protected execution levels and lack a memory-management unit. Consequently, an extended abstraction of the processing architecture, typical for the general-purpose OSs is not needed in the WSN domain. Instead, aspects like the abstraction of platform hardware modules, I/O concurrency management and power management, become prime areas where the developer can benefit from the services of the underlying execution environment.

We use *TinyOS* [88], *MantisOS* [15] and Contiki [42], to illustrate the level of maturity of the hardware abstraction models in WSN execution environments at the beginning of our work on the DASA portability anchor. Our discussion is based on TinyOS 1.15, MantisOS 0.95 and Contiki 0.9.3.

TinyOS

TinyOS is one of the first execution environments specifically designed to meet the requirements of resource-constrained, event-driven and networked embedded systems. Similarly to eCOS, the component-based nature of TinyOS (Section 3.2) allows compile-time customization of the hardware abstraction functionality to the specific needs of each particular application. The component-based model is also instrumental in buffering the changes introduced by the fluid software/hardware boundary by allowing software components to be replaced with real hardware modules (that export the same interfaces) and vice-versa.

The organization of the components that encapsulate the platform hardware modules, however, have been very inflexible and tightly coupled with the specific properties of the mica family of nodes (Section 2.1.2). In Section 4.5 we discuss how the situation was qualitatively changed in the next generation of the OS, TinyOS 2.x, through the application of the decomposition principles of the DASA portability anchor. The description here pertains to the organization of the platform abstraction components in TinyOS 1.x before these changes.

Access to the services of the platform hardware modules can be provided using either *shared* or *virtualized* services. A shared service gives clients full access to the module at the cost of some form of access control. Virtualisation gives each client its own (possibly simplified) “copy” of the module, at the cost of some runtime or latency overhead.

TinyOS 1.x provides one significant virtualized service: the timer. The service offers periodic and one-shot millisecond-resolution events, multiplexed from a single compare register. TinyOS 1.x also provides abstraction for several standard data buses like SPI, UART, I^{2}C, 1-Wire, etc. which are also responsible for arbitration of the shared access. The power control of the peripheral modules is being performed through a standardized interface with a start and stop command.

Most of the differences between the processor architectures in TinyOS 1.x are masked through the use of the nesC/C programming language [65] with a common compiler suite like GNU C Compiler (GCC). The standard C library distributed with the compiler creates the necessary architecture-dependent start-up code like initialization of the global variables, the stack pointer and the interrupt vector table. TinyOS puts the CPU in a low-power state whenever the event/task queue is empty and no interrupts are pending. The selection of the most appropriate sleep mode is computed using chip-specific function that examines the internal registers in the CPU to determine which peripherals are being used at the current time.

MantisOS

MantisOS took a different approach in addressing the specific challenges of the WSN domain than TinyOS. Rather than using a new language and code organization model, it is C based and provides a micro-threaded UNIX-like environment with blocking operations.

The driver architecture in MantisOS closely follows the Portable Operating System Interface (POSIX) model—based on the "everything is a file" abstraction—and supports a small number of system calls with a large number of parameters. For example, the complete interaction between the application code and the driver layer is constrained to only four system calls: *dev_read()* and *dev_write()* for reading and writing data, *dev_ioctl()* for passing device specific configuration information and *dev_mode()* for explicit device power state control.

While the idea of UNIX-like calls might seem appealing initially, the reliability drawbacks outweigh the benefits of comfort. The model pushes the error checking to runtime, as the interfaces do not express the constraints underlying resources. For example, a program can try to read from an sensor using a GPIO pin that does not exist. Consequently, errors that are easily detectable at compile-time under the TinyOS wiring model, can result in run-time errors in MantisOS, and require complicated error detection and mitigation code.

For coordination of simultaneous access, a traditional mutual exclusion approach is used. Each driver maintains a simple "mutex" When the peripheral is locked for exclusive access, any other calling thread is queued in an associated waiting queue and blocked pending the release of the mutex lock by the current owner.

MantisOS drivers that wants to be power managed, must implement a *dev_mode()* function that can be called to modify the power state of the underlying peripheral. Three distinct device power states are supported: on, off and idle. The CPU power management in MantisOS is tightly coupled with the thread scheduling and supports two levels of power saving. When the scheduler ready queue is empty, the scheduler *implicitly* puts the CPU into an *idle* state that consumes less power than the active state, but still supports full peripheral functionality. For greater power savings, the scheduler needs *explicit* information from the threads in order to determine when it is safe for the CPU to go into a *sleep* state. The signaling is performed by a *mos_thread_sleep()* function that threads use to declare the intended duration of sleep. When all threads in the system are sleeping, the scheduler is free to put the CPU into a deeper power-saving state, with only a single timer left running to wake the threads up after the sleeping period is over.

Contiki

The hardware modules are typically accessed in Contiki by calling a particular set of C functions to directly communicate with hardware (e.g., the telosb flash chip, serial port support). In some cases, these functions also communicate with a protothread (Contiki's lightweight, thread-like abstraction for event-based systems [43]) that implements part of the module functionality (e.g., the CC2420 transceiver). Events are signaled by peripherals either by calling a particular function from within an interrupt handler, or by signaling an event to a specific protothread.

There is no general-purpose support for implementing either shared or virtualized services. Some peripherals provide ad-hoc virtualisation (e.g., timer library). Others deal with sharing through various mechanisms: providing only blocking functions (e.g., flash, serial port), synchronization via global variables (e.g., the CC2420 transceiver and access to the I^{2}C bus) and buffering (the TCP/IP networking).

Contiki does not provide any standard mechanisms for managing the power state of peripheral devices. Some peripherals implement on and off functions. Like TinyOS, it puts the CPU in a low-power state whenever the event/task queue is empty and no interrupts are pending.

2.4 Programming Models

In Chapter 1, we motivated the need for a distributed programming abstraction that can enable more rapid development of typical WSN applications. The problem of providing an effective abstraction representing the sensor network services has been a focus of intensive interest in the community. The proposed solutions have ranged from overlay-networks, mobile agents, and application-specific virtual machines to database-like abstractions and publish/interaction schemes. In the following, we briefly review the main properties of these models

2.4.1 Overlay Networks

The increasing availability of high-speed connections at the core but also at the perimeter of the Internet has led to new applications with significantly different characteristics then the traditional client/server model. The so called Peer-to-Peer (P2P) file sharing, previously confined to the LAN environment, became possible on an Internet-wide scale. The limited support from the TCP/IP stack for these new applications was compensated with elaborate application level solutions [72, 93, 153], creating distributed *overlay networks* that run on top of the existing network infrastructure.

The P2P file sharing bears some similarities with the WSNs domain:

- The users are interested in the offered data, and not in the identity of the providers. In the case of the P2P file sharing application this is usually a file with a given name and of a given type. In the sensor networks space this is the sensed data or some other status information.
- The networks can be comprised of many nodes, so scalability and self-organization are required properties.
- The nodes in the network have more or less the same capabilities and act as end-systems and routers at the same time. Each node can be both a provider and a consumer of information.
- The responsibilities of the communication infrastructure are similar: it has to provide means for searching for the data (*lookup*) and then has to provide efficient means for distribution of that data to the interested parties.

Yet the degree of applicability of the solutions from the P2P field into the WSNs space is strongly determined by the actual implementation especially of the lookup and the routing component. The majority of the state-of-the-art solutions in this area are based on the concept of Distributed Hash Tables (DHTs) [173, 178, 190, 224]. In these systems, the required data (e.g. the files), are associated with a *key* that is usually created by hashing (e.g. hash of the filename). A small subset of the total key space is assigned to each node in the network for storage. The overlay network provides set of operations for searching, storing and retrieving data. The `lookup(key)` operation returns the underlying network ID (e.g. the IP address) of the node that stores the data corresponding to the requested key. This then allows the nodes to perform `get` and `put` operations based on the keys.

The main difference between the proposed solutions lies in the routing algorithm that routes the lookup from the source node to the node that most closely matches the associated key value. In a network with n nodes, most of them achieve average path lengths of $O(\log n)$ application-level hops with average $O(\log n)$ neighbors in the overlay address space.

In the above we can detect several properties that can pose as a limiting factors if one would like to build a data-centric framework for WSNs using the same mechanisms.

First of all, the key generation can not be directly applied to the sensed data that is of interest in WSNs. Unlike the file names that are strings, the sensed data is usually represented with real numbers. Directly hashing them into keys would result in a naming interface that is unsuitable for the WSNs applications. The problem arises from the nature of the matching. The look-up process typically performs *exact* matching of the key, thus only answering the equality relation (IS temperature $= 25$). This complicates the realization of interfaces using constraint based naming (e.g. IS temperature > 20), that is very frequently required in the WSN applications. One possible way around this limitation is to sacrifice the flexibility of the naming and create predetermined and fixed *categories* of data, and then create the key as a hash of the name of the category.

From performance point of view, a major weakness is the fact that the overlay is oblivious to the underlying network topology. While the DHT solutions provide good average number of hops in the overlay address space, this can translate into much larger number of hops on the networking level. Also, the potentially large number of neighbors in the overlay address space can translate to a large routing state at each node. Lately, the work on building *topology aware* overlay networks [120, 174, 207] shows promise in overcoming some of the identified limitations. Using geographic routing is another way to increase the coupling between the overlay and the underlying wireless topology. The authors in [175] propose such a Geographic Hash Table (GHT) system as a basis for a Data Centric Storage (DCS) [185] for WSNs. While the DCS approach has the potential to lower the overhead in querying the network (by not requiring a flooding step), the expressiveness of the queries is still limited by the DHT properties, supporting at most category or event granularity.

2.4.2 Active Networks and Mobile Agents

In the active networks model, the users "inject" code in the nodes in order to customize the operation of the network. The high expressive power of the customization code and the fact that it can migrate into the network (in the form of mobile agents) enables the specification of arbitrarily complex in-network manipulation of the data.

This flexibility of the active networks approach can be put to a good use in the WSNs context:

- The possibility to modify or completely replace the software suite of a running WSN is desirable for several reasons. Many of the planned scenarios ask for a long term deployment of the network. During this time it is going to be necessary to implement fixes for the detected software bugs or introduce completely new services in order to support applications that were unforeseen at the time of the initial deployment.

- The mobile agents approach brings a customized processing closer to the data sources, cutting on the number of exchanged data messages. The code can be individually molded for each different application resulting in more efficient applications than using a more general purpose solutions. The efficiency can be further increased if the code is dynamically optimized during its roaming in the network.

But there are also several significant shortcomings that can hinder the active networks model when applied to the WSNs:

- While it offers almost unlimited power of expression, the complete burden of exploiting that flexibility is on the application programmer. One has to provide customized active code for each new application. The middleware as such does not provide much in the form of a general solutions that can be used for rapid development of different applications.

- Although the active code can cut the number of required messages for a given in-network data processing, this comes at a cost for the additional code-distribution messages.

- The mobile agents model can introduce tight temporal coupling between the agents. This complicates the correct operation in the face of link/node failures or other types of transient node unavailability.

- The introduction of active code in the network can increase the security risk, making the network more vulnerable to external and internal attacks.

The first problem can be overcome by providing additional APIs that wrap some of the standard and recurring operations: sensing, networking, mobility, etc. But even with these APIs there is still a trade-off between the programmer efficiency and the efficiency of the solution. The model is best suited for application scenarios that require complex distributed algorithms and dynamic in-network processing.

The authors in [19] present *SensorWare*, an active code architecture based on mobile Tool Command Language (TCL) scripts. These scripts define both the data processing and the replication/migration pattern of the mobile code. The effectiveness of the approach is demonstrated on a target tracking application that relies on complex relations between the nodes participating in the execution of this cooperative task. The use of a scripting language instead of a full-blown compiled language also simplifies the creation of a protected "sandbox" execution environment that can limit the unwanted effects of malicious or misbehaving code.

2.4.3 Virtual Machines

One approach for addressing this code-distribution overhead is the introduction of *virtual machines* on the nodes. These virtual machines can support a customized WSN instructions set that allows quite dense representation of even very large programs and raise the level of abstraction, allowing more rapid application programming.

In [125], the authors present *Maté*, an application-specific virtual machine for TinyOS. The customization of the virtual machine instruction set allows concise representation of the most frequently used WSN operations: reading a sensor, sending a message, turning the Light Emitting Diodes (LEDs) on an off, for example, all take just a single instruction. Similar to the work in [18], additional savings in the code-overhead is achieved by segmentation of the program into smaller building blocks or *capsules* and then implementing smart capsule caching on the nodes. Depending on the application dynamics, the caching of the capsules can significantly reduce the number of required messages. The distribution of the capsules in Maté is preformed via density-aware broadcast mechanism based on the Trickle algorithm [126, 128] that reduce congestion-inflicted capsule loss and by that the settling time for the new program version.

Recently, there has been increased interest [21, 116] in developing optimized virtual machines that share a level of compatibility with the Java Virtual Machine (JVM) [132] and can run on the resource-constrained MCUs typical for the WSN platforms.

2.4.4 Databases

Many of the WSN applications, or at least some of their parts, can be expressed in the form of queries:

- *What is the current average temperature in the building?*
- *What is the maximum humidity in Room 439?*
- *How many sensors in Room 943 detect movement?*

- *What is the average humidity in the areas with temperature higher then 25 degrees?*

This is quite similar to the way the users interact with the traditional databases. The realization has fueled a very active research direction that abstracts WSNs as a database [62, 74, 98].

According to this model, each sensor reading represents a database *tuple*. The combination of these tuples creates an append-only, distributed relational table that can be operated on. The specification of the queries is usually performed using the standard Structured Query Language (SQL). There are some benefits from using this model:

- It provides an application-independent data-centric interface that can increase the efficiency of the application programmer.
- This interface already provides support for selection, grouping and simple aggregation operations, very common tasks in the envisioned WSNs applications.
- It shields the programmer from the volatility of the network, letting him concentrate on the application requirements of the end-user.

As an illustration, the above queries can all be expressed using standard SQL constructs over a virtual *sensors* table [137]:

```
SELECT AVG(temperature)
FROM sensors

SELECT MAX(humidity)
FROM sensors
WHERE location = "Room␣439"

SELECT COUNT(movement)
FROM sensors
WHERE location = "Room␣943"

SELECT AVG(humidity), temperature
FROM sensors
WHERE temperature > 25
```

The simplest and naive way to support the illusion of the WSNs as a database is to send each sensor reading to a central processing node where they are collected in a relational table over which the queries are run. Because of the WSNs limitations, this will seldom be acceptable but for the most simple cases with few nodes. Leaning on previous work in the areas of distributed databases, and the recent developments on operating over "data steams", attempts are being made to specify WSNs optimized in-network implementation of the main database operators [74, 222].

The classical database problem of query-execution optimization is thus transformed into a *routing* problem. The overall success of the model then largely depends on the development of sufficiently efficient and adaptive tuple-routing mechanisms that can support the main database operators, reflecting the specific needs of the

WSN applications. In [137], for example, the authors discuss some of the peculiarities of performing the simple stateless aggregation operations in a WSNs setting. Their aggregation service for the TinyOS architecture runs on top of a tree, routed in the "base-station" that issues the queries. As the values propagate from the leaves back to the root node, they are aggregated according to the aggregation operator in the query.

2.4.5 Publish/Subscribe

The rapid growth of the distributed systems in the last decade has prompted the development of more flexible communication schemes that closely mirror the emerging dynamic and decoupled nature of the applications. The *publish/subscribe* is one such scheme that supports a form of loose interaction between the entities in a distributed system [50]. It is event-based in nature, that is very different from the *request/reply* behavior of the "classical" *point-to-point* and *synchronized* protocols.

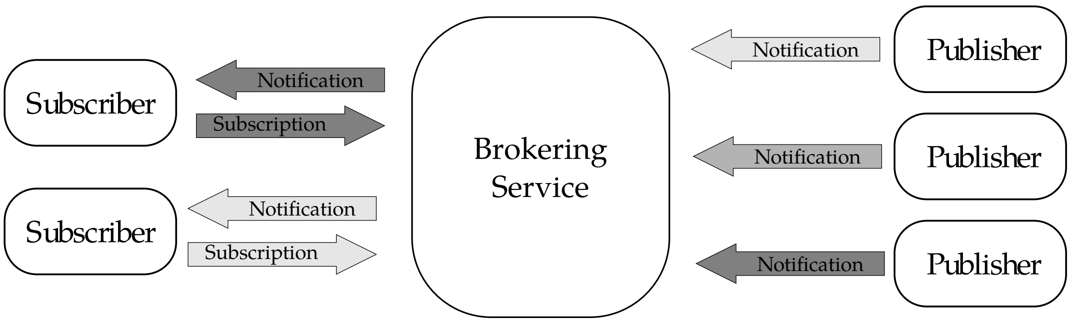


Figure 2.16: The publish/subscribe interaction pattern.

Instead of specifying the identities of the communicating parties, in the publish/subscribe model, the parties specify the nature of the data/events that they are prepared to provide or consume. Figure 2.16 illustrates the interaction pattern. An abstract *brokering* service interposes between the parties and coordinates the interaction, making sure that the matching data is delivered to the interested side. The providers disseminate the data by *publishing* it in the form of notification messages. The receivers declare their interest in certain notifications by subscribing. This is done by sending a *subscribe* message that codifies the nature of the requested data. From this point on, all produced data that is matching the issued subscription is going to be delivered to the subscriber. This continues until the subscriber declares that he is no longer interested in the data by unsubscribing with an *unsubscribe* message.

Identity and Time Decoupling

The publish/subscribe model is well aligned with the service needs of a large class of WSNs applications. In many of them the item of interest is the generated *data* and not the identity of the producer and they benefit from time decoupling between the producers and the consumers of the data.

The *identity decoupling* is the foundation on which the flexibility of the model rests. It represents the fact that the communication between the subscribers and the publishers is performed without knowing the identity of the other side. The whole communication is performed based on the characteristics of the provided or requested *data*. The publishers do not keep references to the subscribers and vice versa. Even the number of the parties participating in the interaction is not known. It can as easily be one-to-one, one-to-many, many-to-one or many-to-many interaction.

This *anonymous* interaction makes the applications built using the publish/subscribe model easily adaptive to the run-time changes in the network. New subscribers can be introduced in the network at any time, and they will start immediately receiving the notifications containing data matching their interest. Similarly, new publishers can be introduced in the system and the existing matching subscribers will start receiving their notifications without any additional readdressing.

In contrast, in the traditional distributed-systems, after the introduction of new services or after changes in the existing ones, the *name service* must be updated. To use the new service, the application has to first contact the name service in order to obtain the identity of the provider and then establish a direct communication. In the publish/subscribe model, as long as the subscribers and the publishers use the same data specification model, there is no need for a naming service, as the data is self-describing.

In addition to being identity oblivious, the publish/subscribe is *time decoupled*, i.e. it does not require that the parties are actively participating in the interaction at the same time. This is to say that a subscribers can subscribe to a given event even when there are no available publishers, and start receiving them once a matching publisher comes on-line. It can also happen, that a subscriber is notified about some data only after the original publisher is disconnected. Similarly, a publisher can start publishing while the subscriber is disconnected, and the subscriber will start receiving the data once it comes on-line.

The same asynchronism is present inside the producers and the subscribers. The publications and the notifications are not usually performed in the main program flow of the application. They are *non-blocking* operations. The publisher is not blocked while producing events, and the subscribers are asynchronously notified when a matching publication is received.

The above decoupling of the production and the consummation of the information increases the scalability by reducing the required coordination between the communicating parties. This makes the publish/subscribe model a suitable framework for implementing large-scale asynchronous applications.

Brokering System

One possible classification of the publish/subscribe systems is depending on the topology of the brokering system that performs the matching between the subscribers and the publishers. Three different variants can be identified:

Centralized In this topology, there is a single entity in the system that acts as a broker for all subscriptions and notifications. When providing data, the publishers first send the notifications to the broker. The broker then matches the publication with the previously received subscriptions and sends the matching parts to the subscribers. This approach is suitable for the applications that require high levels of reliability, data consistency and transactional support. On the other hand, the centralized topology introduces a bottleneck in the system that can severely limit the data throughput. At the same time, the centralized broker is a single point of failure that makes the whole system vulnerable.

Distributed On the other extreme, in the distributed topology, the functionality of the broker service is shared between all entities in the system. Using intelligent store and forward methods, the subscribers and the publishers interact directly while maintaining the appearance of anonymity and asynchrony. Consequently, there is no bottleneck that stands in the way of the scalability. The main shortcomings of this approach is the fact that it requires complex communication primitives that are more error prone that in the centralized approach. As a result, it is much harder to implement higher level services with guaranteed semantics.

Hybrid The hybrid approach tries to combine the characteristics of the both approaches. Instead of using a single dedicated broker system, the broker functionality is distributed among several networked broker servers. The subscribers and the publishers act as clients and associate themselves with a given broker. Each broker server acts as proxy for the subscriptions/notifications of the locally attached clients. The broker network performs a distributed algorithm for matching the various notifications and interests. Many of the properties of the hybrid architecture depend on the topology of this broker network that can be in the form of a tree, acyclic graph or completely peer-to-peer.

Expressiveness

Equally important is the classification of the publish/subscribe systems based on the *expressiveness* of their service interfaces. This has profound influence on the flexibility of the system and limits the type of the applications that can be effectively supported. Currently, three different categories of publish/subscribe systems can be identified based on the type of the *naming* that is used when expressing the interests for the data:

Subject-based This group encompasses the representatives of the earliest publish/ subscribe systems that defined subscriptions based on specific *subjects* or *topics*. It represents a publish/subscribe wrapper around the *group* concept in the multicast communications. Each subject defines one such group. The act of subscribing to a subject is the same as becoming a member to a multicast group, while the publishing an event on a given subject is equivalent to multicasting the event to all members of the group.

Although the tight integration enables straightforward implementation of the middleware (by direct mapping to the multicast support in the network layer), it also results in an inflexible and coarse naming style. The partitioning of the event space using the subject *keywords* has to be performed upfront, and the parties have to agree on this set before they can effectively communicate.

One useful extension is the introduction of *hierarchical* dependencies between the subjects. In this type of systems, subscribing to a node in the subject hierarchy tree also implies the subscription to the events published under the keywords in the subtree. The subject tree provides a mechanism for controlling the coarseness of the subscriptions and in that sense amortizes some of the negative properties, but it the nature of the partitioning remains static.

Type-based The type-based publish/subscribe naming casts the subjects approach in an Object Oriented (OO)-framework. It enables filtering of the notifications based on their *type*. The subject tree is here replaced by an *inheritance graph*. The act of subscribing to notifications of a given type means an automatic subscription the inherited notifications types.

While similar with the hierarchical subjects model, the type-based systems provide some additional conveniences on the programming side. Through a tight integration of the middleware and an OO-programming language, a compile time type checking can be performed, significantly simplifying the debugging of the applications.

Content-based This is the most flexible of the publish/subscribe schemes. It allows the specification of the interests in the form of predicates over the content of the notification message. The service interface of the DASA interoperability anchor leverages this model, and we describe its properties in greater detail in Chapter 5.

CHAPTER 3

Double-anchored Software Architecture

In this chapter we introduce the main features of the Double-Anchored Software Architecture (DASA), focusing on the core organizational principles. We provide arguments for the use of the component framework model as effective vehicle for expressing the architectural constraints, and we argue about the optimal delineation points in the WSN software stack. The two anchors of the architecture are presented in detail in the subsequent Chapter 4 and Chapter 5.

3.1 Cost of Abstractions and Decoupling

In many ways, the existing WSN software development landscape is characterized with inefficiencies similar to the ones that triggered the discussion about the impending "software crisis" in the software engineering community in the 1960s and the ways to combat it through software reuse. Our understanding about the practical relevance of the different reuse techniques has advanced significantly since these early days, and the community focus has turned to broader artifacts of reuse like design patterns and architectural specifications [117]. In Chapter 1 we pointed the important role that common architectural guidelines can play in promoting independent implementation, isolation, and reuse in the WSN software development process. These benefits rely on the specification of effective APIs and abstractions that enable decoupling and provide complexity hiding, as crucial enablers of reuse.

As we aspire to incorporate these classical methods for increasing the software development productivity, we must remain sensitive to the specific constraints in the WSN domain. The decoupling and the complexity hiding increase the portability and provide opportunities for reuse, but come at a price in fidelity and efficiency.

The higher the level of abstraction, the less control one has over the services in the underlying levels, leading to diminished performance in comparison to an integrated solution. As a result, there is a constant tension between the need to raise the level of abstraction as prerequisite for improving portability and productivity, and the need for lean abstractions and vertical integration that maximize fidelity and efficiency.

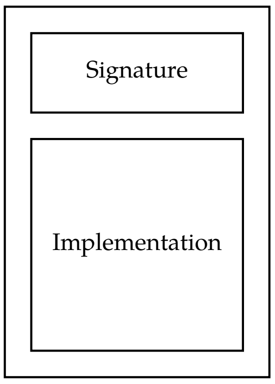
This intrinsic conflict cannot be easily side-stepped. The exponential growth of the hardware capabilities in the general-purpose computing systems has merely enabled this tension to remain relatively well hidden in this domain, due to the hardware over-provisioning. Our hardware survey (Section 2.2) shows that the rate of increase in the capabilities on the WSN platforms is more flat, as the gains from the technological advances are predominantly being applied towards making the nodes smaller and cheaper. Due to this, in the WSN domain, we can not rely solely on the technology to compensate the fidelity and efficiency costs of the software abstractions. On the contrary, balancing between the benefits that abstractions carry and the costs that accompany their use, has to be promoted to a core design goal of the software architecture.

3.2 Component-based Development

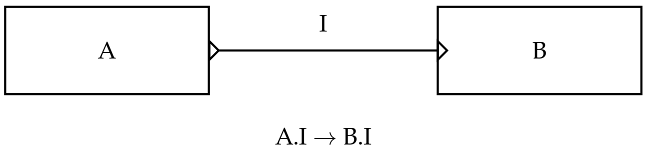
Component modularization is an approach in which the functionality of the traditional monolithic abstraction layers is broken-up in smaller, self-contained building blocks that interact via clearly defined *interfaces* [141]. These building-blocks, called *components*, represent a basic *" unit of composition with contractually specified interfaces and explicit context dependencies only".* [191].

The internal structure of the component cleanly separates between the specification of the service in the component signature, and its implementation (Figure 3.1(a)). This decoupling promotes modularity and reuse, while at the same time allowing rich interaction between the individual building blocks. The services and applications can now be *composed* by "wiring" together the necessary building-blocks (Figure 3.1(b)). This typically entails explicit specification of the involved components, their roles in the interaction (as providers or consumers of services), and the connecting interfaces.

The component-based software model supports rich composition of services. The interaction between the composition units is no longer constrained to a strict up/down direction like in the traditional layered models, but starts to resemble a graph. This enables finer extraction of common functionality and allows reuse across the layers of abstraction, but comes at the cost of more complex dependency relationships. This organizational principle represents a good fit to the requirements of a wide class of embedded systems [61]. Breaking the design into fine, self-contained and richly interacting components provides a viable way for addressing the intrinsic friction between the need for reusability and the cost of abstractions. The flexibility of the model enables the system to better absorb the tension between reusability and efficiency, and to adapt itself to the dynamic hardware/software boundary. These features make the component-based model a suitable tool for breaking the vertically



(a) Component structure: there is clean separation between the service specification in the component signature *and its* implementation



(b) Component composition: component A *accesses services from component* B *via the interface* I

Figure 3.1: Component-based software model

integrated WSN development approach. The model empowers the developers to select the appropriate level of abstraction for the task at hand and to customize the services appropriately.

The flexibility enabled by the component modularization, however, carries a risk of overwhelming the developer with the wide range of options for component selection and composition. Figure 3.2 provides a simplified depiction of the component-based protocol architecture we have developed in the context of the EYES project [51] and illustrates the complexity of the dependency relationships.This creates a strong need for reusable design templates that can impose additional structure on top of the component-based organization and can increase the productivity of the development process.

Many of the existing architectural proposals for WSNs, offering such design templates, have been constrained to the component organization of the communication stack. In [37], for example, following the "narrow waist" approach of the Internet architecture, a new Sensornet Protocol (SP) is proposed as interoperability spanner offering "best-effort, single-hop broadcast" service. In [45], this architecture is further extended with a modular framework for implementing network protocols, NLA, that promotes code reuse and run-time sharing.

We argue that substantial progress in promoting reuse and rapid development is only possible by spreading the structured development approach over wider parts of the software stack, despite the fact that a single architectural framework can not cover the full diversity in WSN applications and hardware platforms.

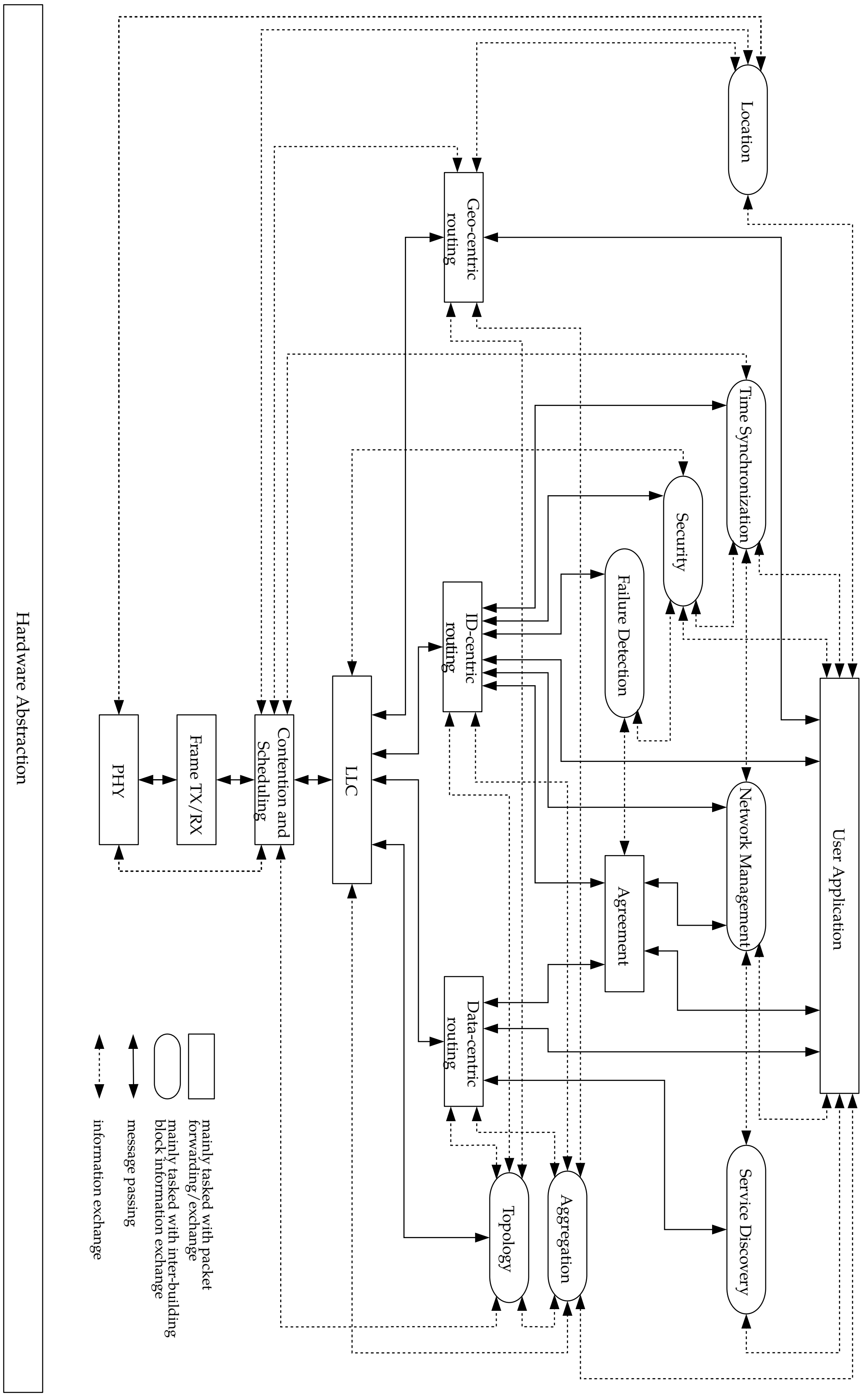


Figure 3.2: Illustration of the rich interactions between the components in an example protocol architecture for WSNs, developed in the context of the EYES project.

3.3 Double-anchored Software Architecture

This dissertation contends that a broad software architecture for WSN, expressed as component framework [107, 124], can strike good balance between rigidity and decoupling—which maximizes opportunities for design and code reuse—and flexibility and configurability—which enable graceful handling of abstraction and decoupling costs.

We propose a DASA, that follows these principles and effectively promotes portability and interoperability while maintaining high sensibility towards abstraction costs. DASA enables adaptive enforcement of design-constraints in different parts of the software stack. The architecture is structured in a set of *anchors*, zones with strong design-constraints and composability restrictions that introduce rigidity and decoupling, and a set of *hot spots* [167], zones with high performance impact and potential for application-specific customization, which remain highly flexible.

The effectiveness of this organization principle crucially depends on identification of salient points in the software stack where fixation of the interfaces and the resulting decoupling brings biggest opportunity for design and code reuse. One approach for detecting the optimal delineation is to find the interfaces that contribute towards better separation of concerns among the different stake-holders in the WSN software development process. Looking at the current software development landscape, we can identify three major classes of WSN developers with specific interests and skill-sets:

Platform developers These developers are intimately familiar with the low-level behavior of a particular hardware platform. They use this knowledge to design and implement support code that abstracts the capabilities of the underlying hardware and forms basis for implementation of higher-level services and applications.

Distributed service developers These developers are experts in communication technologies and distributed systems and are interested in the design and implementation of reusable service building blocks like the communication and sensing stacks, localization and time synchronization protocols.

Application developers These developers are experts for a particular application area like building-automation, asset-tracking, personal health monitoring, etc. They are interested in developing domain-specific applications and are familiar with the application logic, but are not confident with the low-level aspects of the technology. They prefer that all of the complexity of the WSN as a system remains hidden behind a convenient high-level programming abstraction.

The feasibility and the effectiveness of the decoupling between the different stake-holders in the software development process is best exemplified with the apparent success of the *POSIX* [96] and *Berkeley socket* [189] abstractions in decoupling the application from the underlying operating system and networking stack in the general-purpose software. To achieve the goals for more structured software development process we need comparable interfaces in the WSN software architecture that will promote decoupling and reuse, while maintaining the required levels of efficiency. Out of these considerations, DASA defines two anchors of rigidity in the WSN software stack (Figure 3.3):

Portability anchor that abstracts the local services provided by the underlying hardware; and

Interoperability anchor that abstracts the services in remote contexts and allows more rapid development of distributed WSN applications.

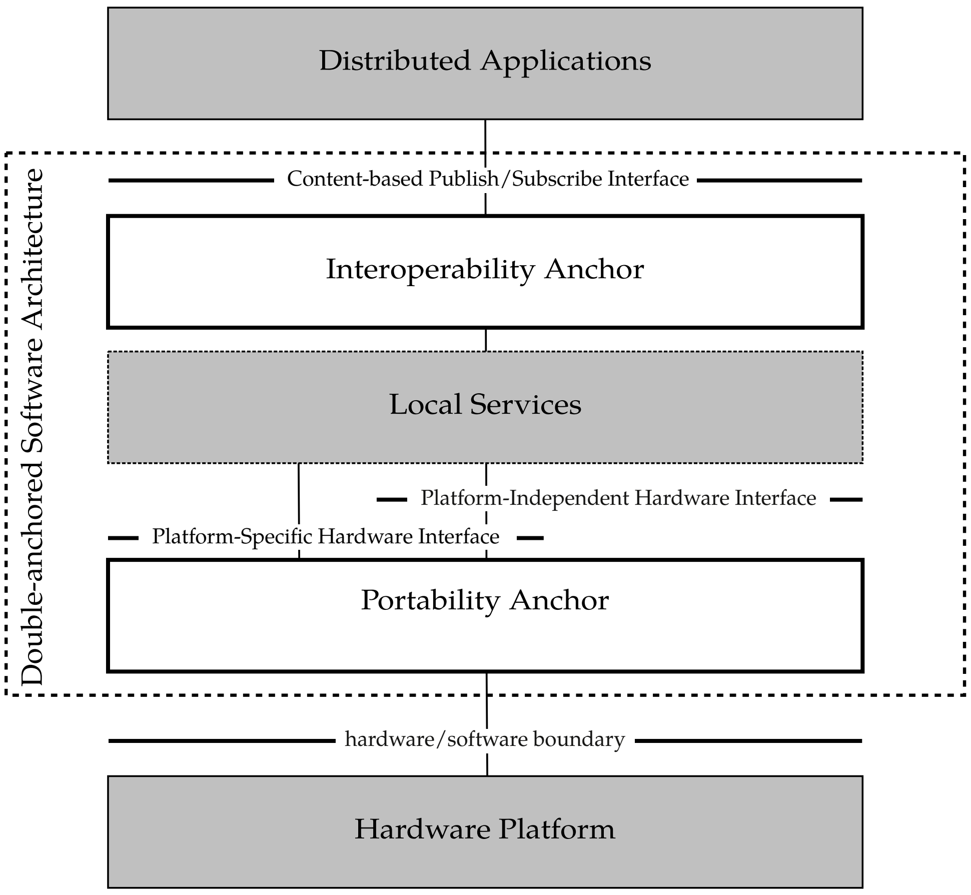

Distributed Applications
Content-based Publish/Subscribe Interface
Double-anchored Software Architecture
Interoperability Anchor
Local Services
Platform-Independent Hardware Interface
Platform-Specific Hardware Interface
Portability Anchor
hardware/software boundary
Hardware Platform

Figure 3.3: High-level functional decomposition of DASA.

In the following we briefly introduce the two anchors and their general features. The detailed specification of their internal organization and the evaluation of the achievement of our design goals is presented in the two subsequent chapters, Chapter 4 and Chapter 5, respectively.

3.3.1 Portability Anchor

Breaking the tight coupling between the software and the underlying hardware through a hardware abstraction is a crucial requirement for amending the inefficiencies of the vertically-integrated development model in WSNs. By hiding the hardware-dependent code from the rest of the system, hardware abstractions play important role in promoting portability and reuse and are thus core element of many execution environments (Section 2.3).

According to the definition in [146], a software artifact is portable *"... to the degree that the effort required to transport and adapt it to a new environment in the class is less than the effort of redevelopment."*. The aim of the portability anchor in DASA is to improve the *source-level portability* of the WSN software, by establishing standard hardware-independent interface to the services provided by the underlying hardware platforms.

In this, we are following a similar goal as the POSIX standard [96] which *"... specifies Application Programming Interfaces (APIs) at the source level, and is about source code portability. Its neither a code implementation nor an operating system, but a stable definition of a programming interface that those systems supporting the specification guarantee to provide to the application programmer"*. The POSIX approach of narrow interfaces and monolithic abstractions, however, is not suitable for direct application in the WSN domain.

As discussed in Section 3.1, in the WSN context, the cost of abstractions cannot be as easily masked by hardware over-provisioning as in traditional systems. Consequently, mechanisms are needed for avoiding some of the abstraction overhead in cases when the need for performance overshadows the benefits of the complexity hiding. The hardware abstraction in WSN has to be more modular. It has to accommodate a fluid hardware/software boundary and it has to delineate between the development of complexity hiding abstractions and their equalization across different platforms.

The portability anchor in DASA codifies these design-constraints that we deem necessary for effective organization of software along the hardware/software boundary. The anchor is structured as a three-level component framework that progressively abstracts the capabilities of the underlying hardware platform. The top-level components offer public *hardware-independent interfaces* for building portable services and applications, and the middle-level components offer public *hardware-specific interfaces* which provide access to the full capabilities of the underlying hardware.

This organization of the hardware abstraction functionality provides a solid foundation for developing hardware-independent services and applications, allowing significant code reuse across different hardware platforms. At the same time, the design offers mechanisms for flexible control of the performance penalty for this portability. In situations where the performance loss is too high, the developer can skip the portability abstraction and directly tap to the hardware-specific interfaces.

3.3.2 Interoperability Anchor

The lower anchor in DASA decouples the evolution of the software from the underlying hardware but does not provide sufficient complexity hiding for significant gains in productivity of the development process. To enable more rapid development of distributed WSN applications, a higher-level *interoperability spanner* is needed that will protect the developer from the complexity of accessing services both in the local and remote execution context.

In many WSN application domains, a classical networking API like the Berkeley socket does not provide adequate levels of abstraction for substantial productivity gains. Unlike the identity-centric, any-to-any communication patterns, characteristic for the traditional communication networks, the focus in WSNs is on the sensed data and its context, and less on the identity of the node that acquired it. This creates a need for customized communication abstractions that can better support the specific interaction patterns of the WSN applications. Many of the existing WSN standardization activities like ZigBee [225], WirelessHART [217], and ISA100 [101], try to address this demand through vertically integrated service architectures that have narrow scope and lack flexibility and extensibility.

The interoperability anchor in DASA offers a more generic and flexible approach. It exports a light data-centric abstraction that shields the application from the evolution of the underlying service code and provides a sufficient complexity hiding enabling rapid development of distributed WSN applications. The anchor is organized as component framework with a service interface that follows the Content-Based Publish/Subscribe (CBPS) model, tailored to the specific requirements of the WSN domain.

The framework fully decouples the implementation of the service API from the *hot spots* in the architecture, like the communication and the sensing stacks, that have high impact on the application performance. From one side, this promotes independent evolution of the solutions below the anchor. From the other side, the decoupling enables easy application-specific customization of the service abstraction through the use of different communication substrates and extension components. For example, by choosing between a flooding protocol and a dissemination protocol [130, 198], the application developer can select between reliable and unreliable distribution of the subscriptions.

3.3.3 Configurability

Our architectural proposal is tailored to the specific challenges associated with software development for resource constrained WSN platforms. The presented component-based organization and the decoupling constraints are enforced only at the source-level organization and the architecture explicitly allows collapsing of the decoupling barriers and interface optimization during the code compilation process. In this way, the additional structure in the software stack is maintained only during the development phases where it generates maximal benefits in terms of reuse and improved productivity.

In contrast to run-time component frameworks like RUNES [156], we believe that for the class of resource constrained WSN devices, the memory and the invocation overheads associated with maintaining full run-time component-based organization negate the gains resulting from the increased flexibility. The ability to dynamically exchange components at run-time also has to be weighed against the potential for introducing subtle composability problems which are expensive to detect and fix.

In line with this view, DASA does not provide rich mechanisms for run-time adaptation like introspection and reflection, and lacks dedicated cross-layer data sharing services [115, 121]. Instead, DASA relies on explicit exchange of metadata—comprised of simple attribute/value pairs—as versatile run-time control signaling mechanism both "vertically", among components in a local execution context, as well as "horizontally", across different WSN nodes and contexts (Section 5.4.2).

3.4 Related Work

The ARPA project on Domain-Specific Software Architecture (DSSA) [1] in the 1990's has marked the start of increased interest in the software engineering community in developing reference architectures for specific application domains. This interest has lead to designs like IBM's ADAGE reference model for avionics [8] or Philips' Koala model for consumer electronics [203]. More recently, a group of leading automotive manufacturers and suppliers has initiated AUTOSAR [86], a joint initiative for developing an open reference architecture for automotive electronics systems that is agnostic from the implementation language and the execution environment.

As a broad domain-specific architectural proposal, DASA shares many high-level goals with initiatives like AUTOSAR. Both attempt to address the existing complexity and inefficiencies in the software development process through the identification of core abstractions, standardization of interfaces and interaction patterns. Despite the fact that both initiatives target resource constrained embedded devices, the different application contexts have led to significantly different architectural decisions. These differences highlight the adaptation of DASA to the specific needs of the WSN domain.

The "Basic Software" in AUTOSAR serves the same role as the DASA portability anchor. It abstracts the rest of the system from the differences in the MCU architectures and offers common device-driver API. Unlike DASA, however, the abstraction is monolithic and does not allow flexible selection of the portability/efficiency trade-offs. Another crucial difference is in the scope of the architecture. The AUTOSAR "basic software" incorporates a full specification of the execution model [159], while DASA is OS agnostic and does not impose strong constraints on the concurrency model.

The interoperability anchor in DASA shares many design goals with reconfigurable middleware solutions like the Y distributed application platform [165], Gridkit [75] and BASE [10]. In contrast to these approaches, DASA is optimized for hardware platforms that have orders of magnitude less available resources and is thus much less general. Although we share the same goal of decoupling between the implementation of the service and the underlying communication substrate, DASA does not support different transactional models or different concurrency classes in the service invocation. Like BASE, the component framework of the DASA interoperability anchor follows the micro-broker approach, and allows extensibility with plug-in components. However, due to the resource constrained nature of the WSN platforms, we focus on compile-time configuration supported by simple metadata-based control signaling at run-time.

CHAPTER 4

Portability Anchor

In this chapter we present the low-level foundation of DASA that paves way for portable development through progressive abstraction of the resources on the underlying hardware platform.

The architecture of this *portability anchor* is based on a set of vertical and horizontal composability rules that augment the generic component-based organization model and allow for flexible organization of the hardware abstraction code. The resulting component framework provides efficient support for building platform-independent services and applications, while still allowing direct access to the full features of the underlying hardware, when the need for performance and efficiency outweigh the need for portability.

We conclude the chapter with an evaluation of the level of achievement of the main design goals through micro-benchmarking and analysis of the application of the proposed component framework in the code-base of TinyOS 2.x, a popular execution environment for WSN.

4.1 Design Goals

The existence of a hardware abstraction layer in traditional software architectures has proved to be an indispensable method for simplifying application development and for breaking the tight vertically-integrated development method.

The first goal of a hardware abstraction layer is to shield the rest of the system from the intricacies associated with low-level hardware access through wrapping of the hardware resources into more convenient higher-level abstractions. This complexity hiding reduces the cognitive burden when developing higher-level services and contributes towards safe sharing of the available hardware resources.

The hardware abstraction layer achieves its second goal through equalization of these abstractions across the different supported platforms. Through this, it provides basis for developing portable services and applications by defining a platform-independent API against which portable code can be written [146]. This portability barrier decouples the higher-level services from the evolution of the underlying hardware as long as the platform-independent API can be realized on each new platform. In this way the overhead of porting N applications on M new platforms is reduced from $\mathcal{O}(N \times M)$ to $\mathcal{O}(N + M)$.

Existing portability frameworks, like the ones reviewed in Section 2.3.1, frequently conflate these two aspects of the hardware abstraction functionality. Their interfaces are narrow, rigid and expose the capabilities of the hardware at a single level of abstraction. This unavoidably leads to suboptimal utilization of the available resources when developing platform-specific services and applications. While the portability penalty might be acceptable for hardware over-provisioned systems, the resource constrained nature of WSNs requires a more flexible approach that offers streamlined access to the hardware when necessary.

Ins-trad in our view, a successful hardware abstraction layer for WSNs must enable:

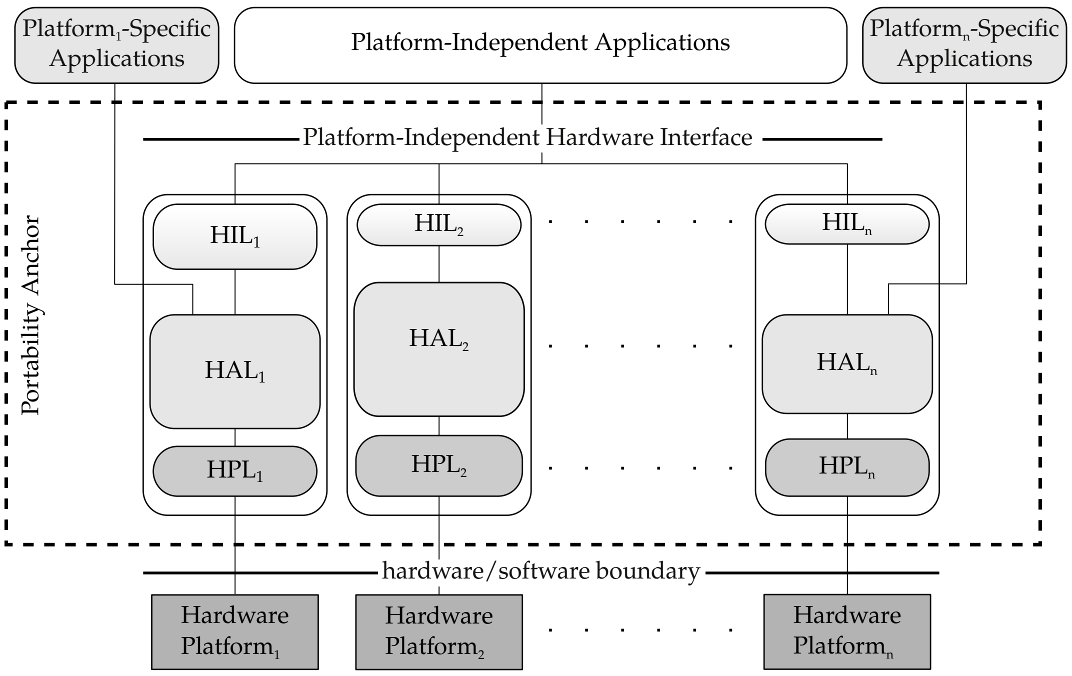
- creation of effective abstractions for the hardware resources of the underlying platform;
- development of portable applications that can run with minimal change on different hardware platforms; and
- effective control over the portability/performance trade-offs.

To achieve these goals, the hardware abstraction in WSN has to be modular: it has to accommodate a fluid hardware/software boundary and it has to delineate between the development of complexity hiding abstractions and their equalization across different platforms.

We argue that these challenges can be effectively met through the introduction of a *portability anchor* in the software architecture that combines the flexibility of the component-based model with a set of *vertical* and *horizontal* decomposition guidelines which we present in detail in the next sections. The combination of these decomposition guidelines with the component-based organization results in a component framework for organizing the hardware abstraction functionality in WSNs that strikes a fine balance between the conflicting needs for complexity hiding and portability, from one side, and system efficiency from the other.

4.2 Vertical Decomposition

To achieve the above goals, we propose that the hardware abstraction functionality should be organized in three distinct layers of abstraction components. Each layer in this decomposition has clearly defined responsibilities and is dependent on interfaces provided by the lower layers.



Platform₁-Specific Applications
Platform-Independent Applications
Platformₙ-Specific Applications
Portability Anchor
Platform-Independent Hardware Interface
HIL₁
HIL₂
HILₙ
HAL₁
HAL₂
HALₙ
HPL₁
HPL₂
HPLₙ
hardware/software boundary
Hardware Platform₁
Hardware Platform₂
Hardware Platformₙ

Figure 4.1: Vertical decomposition of the portability anchor

Figure 4.1 depicts the proposed component framework. For each hardware platform the hardware abstraction functionality is organized in three layers of components. Sitting on top of the hardware/software boundary, the Hardware Presentation Layer (HPL) structures access to hardware registers and interrupts, the Hardware Adaptation Layer (HAL) promotes efficiency through rich hardware-specific abstractions, while the Hardware Interface Layer (HIL) fosters portability across the abstracted platforms by exporting a platform-independent hardware interface.

Through this gradual abstraction of the capabilities of the underlying hardware, on each layer the components become less and less hardware dependent, giving the developer more opportunity for designing and implementing reusable services and applications.

The separation between the hardware-specific abstractions in the HAL and the portability wrappers in the HIL is a core feature of the proposed architecture and its main mechanism for resolving the implicit conflict between the need for portability and the price associated with raising the level of abstraction.

When there is a need for maximal performance, the architecture facilitates writing efficient platform-specific code by offering public access to HAL interfaces, effectively circumventing the portability penalty associated with HIL components.

Thanks to the component-based nature, the portability anchor also supports flexible mixing of these two public interfaces. The higher-level code can be organized in such a way that the majority of the implementation remains platform-independent while only the performance critical parts tap into the platform-dependent HAL interfaces. In spirit, this is similar to the possibility of using inline assembly for implementing the most performance sensitive operations in a C program, or the ability to extend interpreted languages like Python with C/C++ routines.

In the following we describe in more detail the core design principles for each of the component layers comprising the DASA portability anchor.

4.2.1 Hardware Presentation Layer

The components belonging to the Hardware Presentation Layer (HPL) are positioned directly over the hardware/software boundary. As the name suggests, their major task is to "present" the capabilities of the hardware using the native concepts of the component-based organization model. They access the hardware in the usual way, either by memory or port-mapped I/O. In the reverse direction, the hardware can request servicing by raising interrupts. The goal of the HPL is to wrap this low-level interaction with the hardware in the form of components that export more structured interfaces to the rest of the system.

The HPL components are *stateless* and expose interfaces that are fully determined by the capabilities of the hardware module that is abstracted. This tight coupling with the hardware leaves little freedom in the design and the implementation of the components. Even though the actual interface signatures of the HPL components will be as unique as the underlying hardware, having a common general structure simplifies the interaction with the rest of the architecture. To this aim, each HPL component should provide:

- commands for initialization, starting, and stopping of the hardware module, necessary for effective power management;
- "get" and "set" commands for the register(s) that control the operation of the hardware;
- commands with descriptive names for the most frequently used flag-setting and flag-testing operations;
- Interrupt Service Routines (ISRs) for the interrupts generated by the hardware module; and
- commands for enabling and disabling these interrupts.

In addition to facilitating better integration with the rest of the architecture, these interface guidelines also bring benefits for the programmer: instead of cryptic macros and register names whose definitions are hidden deep in the header files of compiler libraries, the low-level services of the hardware module are now directly exposed in the interface signature of the HPL component, making their use much more convenient and less error-prone.

In terms of complexity hiding, the HPL components do not provide any substantial abstraction beyond automating frequently used command sequences. The interrupt service routines in the HPL components perform only the most time critical operations (like copying a single value, clearing some flags, etc.), and delegate the rest of the processing to the higher level components that have extended knowledge about the state of the system.

Despite this, by hiding the most hardware-dependent code, the HPL enables easier development of higher-level abstraction components which can be reused with different hardware-modules of the same class. For example, many of the MCUs used on the existing WSN platforms have two USART modules for serial communication. They have the same functionality but are accessed using slightly different register names and generate different interrupt vectors. The HPL components can hide these small differences behind a consistent interface, making the higher-level abstractions resource independent. The programmer can then alternate between the different Universal Synchronous Asynchronous Receiver Transmitter (USART) modules by simple switching of the HPL components, without any changes to the implementation of the higher-level services.

4.2.2 Hardware Abstraction Layer

The Hardware Adaptation Layer (HAL) components form the core of the portability anchor. They use the raw interfaces provided by the HPL components to build powerful high-level abstractions that mask the intricacies associated with the use of hardware resources. In contrast to the HPL components, they are free to maintain internal state for implementing their services and for performing access arbitration and resource control (Section 4.4).

HAL abstractions offer effective and convenient access to the full capabilities of the underlying hardware, without compromises in favor of increased portability. This goal has parallels with the Exokernel concept that calls for banishing heavy hardware abstractions from the OS and their transformation into libraries that offer finer control over the abstraction penalty [46]. The complexity hiding in the HAL is tailored to the specific features of the abstracted hardware module. Instead of hiding the individual features of the hardware class behind generic models, HAL components provide the "best" possible abstraction that simplifies the development of higher-level code while maintaining effective use of resources.

This customized abstraction approach of the HAL layer is reflected in the exported interfaces. Rather than using a generic "everything-is-a-file" model for all devices—the approach that is used in the POSIX-inspired hardware abstraction architectures [15, 23, 48]—we propose that the HAL components should provide access to these services via rich interfaces built on top of domain-specific models like *Alarm* (Section 4.5.3), *ADC*, *EEPROM*, etc. In [82] we provide several examples of such domain-specific models and illustrate how they can be effectively used to encapsulate the capabilities of the major hardware modules on a typical WSN platform.

At a glance, the narrow-interface approach of the POSIX model seems more convenient to the developer. The functionality of each hardware device is hidden behind a small generic set of commands like *dev_read()*, *dev_write()*, *dev_mode()*, *dev_ioctl()*. This sense of simplicity, however, is misleading: all important aspects of the call are hidden in the command arguments, often in non-transparent pointers to configuration structures that are hard to test for correctness, especially on mote-class devices. In contrast, the use of rich, expressive domain-specific interfaces enables implementation of stronger compile-time invocation checks, which increases the robustness of the code in light of complex component composition.

4.2.3 Hardware Interface Layer

The final tier in our architecture is formed by the Hardware Interface Layer (HIL) components that take the hardware-specific abstractions provided by the HAL and convert them to hardware-independent interfaces used for writing portable code. In effect, the HIL components encapsulate the "portability penalty" of the hardware abstraction. The overhead contained in the HIL components represents the additional price, relative to the HAL, that the application has to pay to be able to run on different platforms.

As a result, the relationship between the level of abstraction expressed by the common HIL interface with respect to the capabilities of the platforms whose differences it needs to hide, poses as crucial design decision. A typical, but counterproductive option, is to anchor this interface at the Least Common Denominator feature set, which is readily provided by all of the abstracted platforms. This approach does not require significant upfront costs associated with developing portability wrappers, but needlessly constrains the performance of the system on all but the least capable platform, since the platform-independent code can not benefit from the increased function set and performance offered by the more advanced platforms.

In our view, this platform-independent API "contract" has to track the *typical* set of hardware services that are needed for effective implementation of the current state-of-the-art services and applications. Consequently, the complexity of the HIL components mainly depends on the gap between the capabilities of the abstracted hardware and the established platform-independent interface. When the capabilities of the hardware exceed the current API contract, the HIL must "downgrade" the platform-specific abstractions provided by the HAL until they are leveled-off with the chosen standard interface. Correspondingly, when the underlying hardware is inferior, the HIL has to resort to software emulation of the missing capabilities.

As newer and more capable platforms are introduced in the system, the pressure to break the current API contract will increase. When the performance requirements outweigh the benefits of the stable interface, a discrete jump can be made that realigns the platform-independent API. This will force a reimplementation of the affected HIL components. On one side, for newer platforms, the new HIL will be much simpler because the new API contract and their HAL abstractions will be better aligned. On the other side, the cost of boosting up (in software) the capabilities of the old platforms will increase.

The periodic realignment of the platform-independent interface can lead to compatibility issues in long-lived deployments. The standard solution to this problem is to associate a unique version number to each iteration of an HIL interface. This *versioning* of the HIL interface allows applications to safely reference legacy interfaces and maintain compatibility with previously deployed devices.

4.3 Horizontal Decomposition

As affirmed by our extensive survey in Section 2.2, the majority of WSN platforms are built out of COTS and there is a substantial reuse of components across the different platforms. This fact should be reflected in the organization of the hardware abstraction functionality in order to promote reuse of those abstractions which are common on different platforms. Thus, in addition to the vertical decomposition presented in the previous section, the abstraction code needs horizontal modularity that tightly follows this platform composition.

The COTS components used on the WSN platforms use well-defined physical interfaces like GPIO, UART, SPI, etc. By reflecting these physical interfaces into software as platform-independent interconnection abstractions, it becomes possible to reuse the abstractions corresponding to the shared chips across different platforms, which significantly reduces the porting overhead.

The horizontal decomposition using platform-independent interconnection abstractions is also a prerequisite for establishing more decentralized device driver development model in the WSN domain, one that can mirror the established practices for Personal Computer (PC)-class systems where hardware manufacturers develop their own drivers against a well-defined API supported by the execution environment [118].

In the following we describe in more detail the three main concepts: *chips*, *platforms* and *interconnect* that the portability anchor uses to achieve the required horizontal modularity.

4.3.1 Chips and Platforms

Chips are self-contained abstractions of a particular "hardware chip" (MCU, transceiver, flash-chip, etc.). Each chip abstraction follows the vertical decomposition principle defined in Section 4.2, providing both a public chip-specific HAL interface, as well as a public chip-independent HIL interface. Consequently, a *platform* abstracts a particular "hardware platform", and is built as composition of platform-independent chip components which are connected together using platform-specific "glue" components that perform the necessary interface mapping and configuration.

Figure 4.2 illustrates the relationship between these two core concepts of the horizontal decomposition. It shows the organization of the abstraction functionality of a WSN platform as a collection of platform-independent, chip-specific abstractions

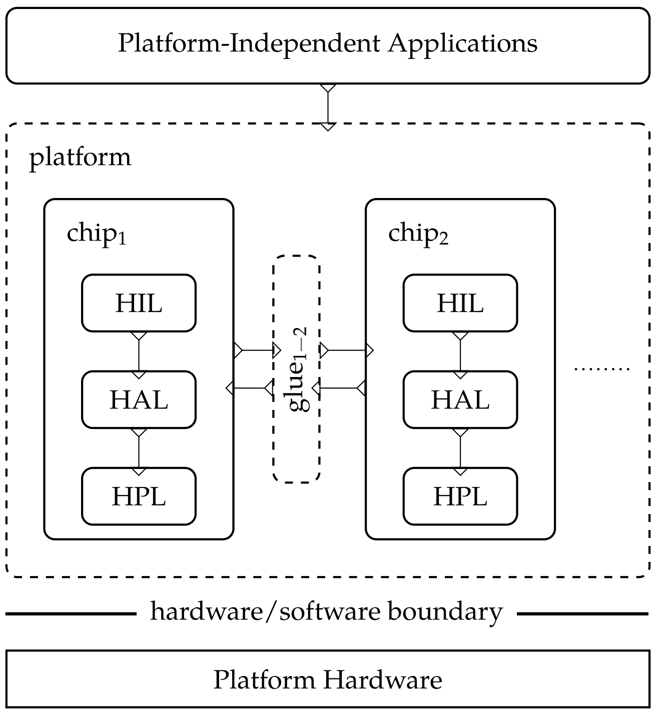


Figure 4.2: Horizontal decomposition of the portability anchor in chips *and* platforms*.*

(chip$_1$, chip$_1$,...) that are interfaced together with the help of platform-specific glue components (glue$_{1-2}$,...).

Thanks to their platform-independent nature, the chip abstractions are reusable across different platforms, resulting in a much more efficient platform porting process. In Section 4.5.4, we illustrate in more detail this decomposition and the resulting portability gains, using the interfacing between the CC2420 and MSP430 chips on the telosb platform as an example.

4.3.2 Interconnect

Figure 4.2 also highlights the central role that the abstraction of the interconnect (Section 2.1.1) plays in enabling reuse of chip-specific code across different platforms.

Our architecture would support greatest portability and reuse by following a generic interconnect model like the one in NetBSD that abstracts the different interconnection protocols under one common memory access scheme (Section 2.3.1). In this approach the abstraction of the chip is completely decoupled from the abstraction of the interconnect, potentially allowing the same chip to be used with different connection protocols on different platforms. This generalization, however, is associated with notable performance penalties due to the involved access translations. Although this additional overhead may be acceptable on more capable hardware, it is highly undesirable in the case of the resource-constrained WSN platforms.

We believe that adopting a less generic approach offers more favorable trade-offs.

Consequently, we have opted for maintaining separate abstractions for the main interconnect protocols. These distinct interconnect abstractions offer possibility for protocol-specific optimizations: the SPI bus abstraction (Figure 2.3), for example, does not have to incorporate the concept of client addresses, while the I^2C abstraction must do so.

The platform-independent abstraction of these interconnection protocols enables easy reuse of chips on any platform as long as it provides support for the required interconnection protocol. For example, the CC2420 chip abstraction can be reused both on the telos and micaz platforms because the abstractions of the serial modules on the MSP430 and Atmega128 MCUs support common SPI interface (Section 4.5.4). One potential opportunity for reuse that is not optimally handled by our approach relates to SoC designs where the SoC chip incorporates a sub-element that is also available as a stand-alone chip. For example, the integrated transceiver in the Texas Instruments CC2430 SoC is very similar to the stand-alone CC2420 design. In this case, direct reuse of the stand-alone CC2420 chip abstraction for easy composition of a new CC2430 chip abstraction is not possible because the CC2420 abstraction references the SPI abstraction as specific interconnect protocol [9].

The clean abstraction of the interconnect is also instrumental for structuring the development of hardware abstraction code in separate domains of expertise and enabling more streamlined device driver development process. For example, the separation between the abstraction for the ADC module and the one for the connected sensors opens way for platform-independent development of sensor drivers. The platform-independent nature of the ADC HIL allows a sensor device-driver developer to code a sensor transfer function in his driver code, without knowing the exact nature of the sensor connection on a given platform (channel number, ratiometric or absolute configuration, etc.), parameters that are separately provided by a platform integrator in the platform-specific "glue" components.

4.4 Concurrency and Power Management

Following the vertical decomposition principles presented in Section 4.2, the services of each hardware resource in the system can be accessed through two public APIs: at the hardware-independent HIL, for portable code, and at the hardware-specific HAL, for high-performance code. As a result, the code above the portability anchor can attempt concurrent access to an underlying hardware resource from different call-sites and using different levels of abstraction. Similar concurrency issues also arise in the context of the horizontal decomposition when multiple chips are connected to the same interconnect. For example, the CC2420 transceiver is connected to a dedicated SPI bus on the micaz platform, but on the telosb platform it shares the SPI bus with the external storage chip, leading to potential access conflicts.

To handle these types of access concurrency, the portability anchor mandates the inclusion of proper arbitration and resource control functionality in the HAL components. Before accessing a potentially shared resource, the client component has to ask and be granted exclusive access by interacting with a special *Arbiter* component that acts as concurrency lock. Each shared resource is protected by a separate arbiter which coordinates the access among the set of client components that have preregistered with the arbiter's interface as candidate lock holders.

Due to their sentinel role, the arbiter components have direct insight into the I/O access patterns in the system. This valuable information can be leveraged to build more advanced higher level services. This observation has led us to propose the use of *power locks*, a novel synchronization primitives that couple concurrency, power management and hardware configuration [113].

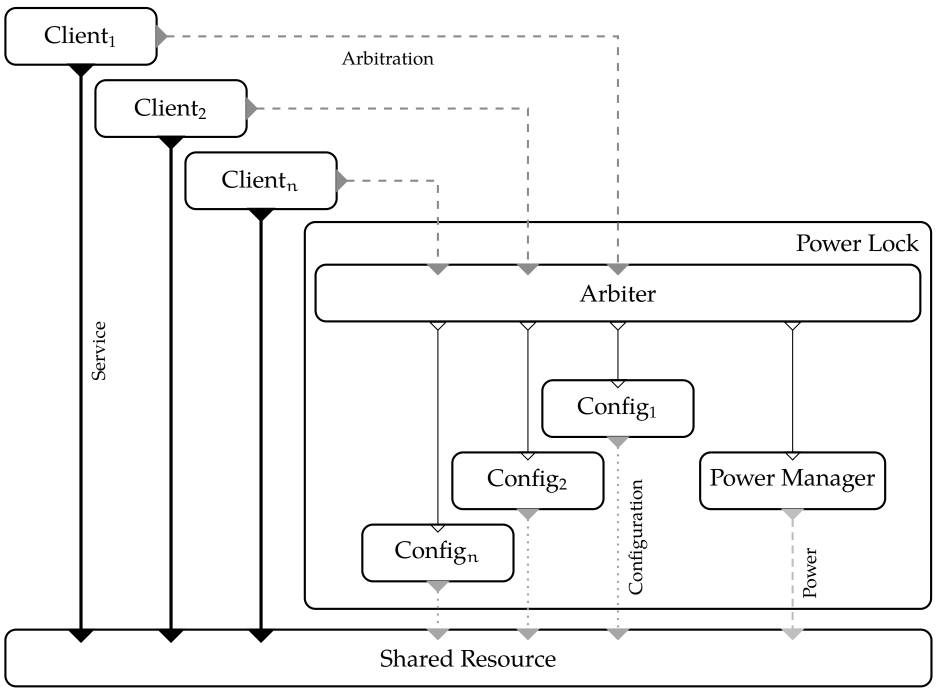


Figure 4.3: Integrated management of concurrency, configuration and power using power locks.

Figure 4.3 depicts the internal structure of a power lock and its external interfacing. It is comprised out of three types of components: an *Arbiter*, responsible for access arbitration, *Configurators* responsible for client-specific configuration, and a *Power Manager* responsible for enforcing a specific power management policy.

Before using the services of the *Shared Resource* through a resource-specific set of service interfaces, the clients (*Client*$_1$, *Client*$_1$,..., *Client*$_n$) have to first acquire exclusive access by sending a corresponding request to the power lock's *Arbiter*. The Arbiter queues the client's requests and grants access to the shared resource according to some specific granting policy (FIFO, round-robin, etc.). As part of the access granting path, the Arbiter calls an optional *Config* component which encapsulates all the client-specific configuration that is needed at each grant event.

The *Power Manager* component acts as a *default owner* of the lock and receives exclusive access to the underlying resource whenever the lock is idle and there are no pending requests from the clients. When holding the lock, the Power Manager can enforce a specific power management policy for the resource through the explicit power management interface offered by the resource (Section 4.2.1). In the most simple case, as soon as the lock goes idle and the Power Manager get access to the resource, it can use the explicit power management interface to immediately power down the hardware module. Consequently, on receiving an access request from any of the clients, the Arbiter informs the Power Manager that the resource needs to be used. The Power Manager powers up the hardware module before releasing the lock back to the Arbiter who runs the client-specific configuration in the Config component before granting access to the client.

In this way even applications with no explicit energy management can operate at energy-efficiency levels very close to a hand-tuned solution. The energy management policy of the Power Manager can be either preselected at compile time or modified at run-time, through integration with a more comprehensive framework for node-level or network-wide dynamic energy management like the ones presented in [11, 105, 122, 209], offering even larger savings.

From the point of view of the client components, the additional configuration and power management services provided by the power lock are visible only as increase in the latency of the access granting process. This additional delay is primarily determined by the time overhead of the power-state switching in the underlying hardware module. In cases where this overhead is significant, the Power Manager can apply more sophisticated management policies, like keeping the module powered on until some timeout, in expectation of an early new client access request.

In some situations, such uncertainly in the latency of the granting path might be unacceptable for the client. To support these use-cases, the power lock also offers a separate, streamlined access requesting interface which either *immediately* grants access to the resource, or returns an error code informing the client that such low-latency arbitration is not possible, allowing the client to take mitigating actions, like retrying at later more convenient and for the client deterministic time.

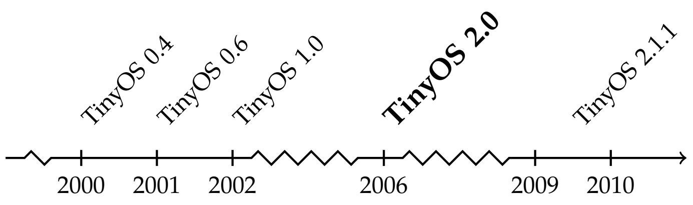
4.5 Implementation in TinyOS 2.x

The early releases of TinyOS 1.x were lacking a clear organization of the hardware abstraction components and the exported hardware abstraction interfaces were strongly biased by the features of the Atmel ATmega MCUs used on the mica family of nodes (Section 2.3.2). This has hampered porting to new platforms and was deemed as one of the most serious shortcomings preventing wider adoption of the OS.

The situation was somewhat improved when we introduced the *msp430* platform in TinyOS 1.1.7, that abstracted the capabilities of the Texas Instruments MSP430 MCU family. This new platform, used as basis for the TinyOS 1.x telos and eyesIFX ports, featured a novel hardware abstraction approach with gradual formation of platform-independent interfaces [81].

The lessons that we have learned through these early porting efforts have lead us to the specification of a three-layer hardware abstraction architecture [82] which became a cornerstone for TinyOS 2.x [127], the new version of the operating system.

TinyOS 2.x maintains many of the basic concepts of its successful predecessor TinyOS 1.x (Figure 4.4), but extends the design in key areas like greater portability, improved robustness and reliability.



TinyOS 0.4	Initial public release by UC Berkeley.
TinyOS 1.0	First version using the nesC language.
TinyOS 2.0	Complete rewrite, TinyOS Alliance formed.
TinyOS 2.1.1	Latest release.

Figure 4.4: TinyOS release history and development milestones.

The development process in TinyOS 2.x is organized around "Working Groups" that focus on different design and engineering aspects with the goal of improving and adding new services to the OS. The organization of the hardware abstraction functionality and the reference implementations fall in the responsibility area of the TinyOS 2.x Core Working Group [197]. Through the activities of the Core WG, we have refined our original three-layer hardware abstraction architecture with a horizontal dimension, and have added more advanced concepts like power locks, resulting in the architecture presented in this dissertation.

In the following, we first provide a briefly review of the other novelties in TinyOS 2.x with respect to TinyOS 1.x, before turning our attention to the implementation of the portability anchor in the TinyOS 2.x code-base.

4.5.1 General Features of TinyOS 2.x

TinyOS 2.x is based on a new version of nesC [66], which extends the original component model used in TinyOS 1.x with the concept of *generic components* and *generic interfaces*. Similar to classes in object-oriented programming, a generic component can be used to create multiple component instances. The instances can be customized using a list of parameters that can also include type arguments. All generics are instantiated at compile-time and the instances are completely independent copies, similar to the way C++ templates work.

The new nesC version also improves the networking interoperability of TinyOS 2.x code, through the definition of a *network type* at the language level. Using this new concept, programs can declare structures and primitive types that follow a cross-platform (1-byte aligned, big-endian) layout and encoding. This allows services to specify platform-independent packet formats without resorting to macros or explicit serialization [27].

In addition to the underlying language changes, TinyOS 2.x introduces a new task posting semantics. In the original TinyOS 1.x design, all components share a fixed-size task queue and a given task can be posted multiple times. This causes a wide range of robustness problems because if a component is unable to post a task, due to the queue being full, it may cause the whole system to hang. To mitigate this problem, in TinyOS 2.x, every task has its own reserved slot in the queue and can be posted *only once*. The new semantics leads to simplified code and more robust components.

This approach of compile-time allocation and binding is applied to all aspects of the system: components preallocate all the state they might possibly need at compile-time and important invariants are explicitly reflected through interface signatures, rather then through parameters that need run-time checking. This design principle limits the flexibility, but makes many of the OS behaviors more deterministic.

4.5.2 Portability Anchor's Implementation

Like other mandatory design specifications, the hardware abstraction architecture in TinyOS 2.x is described in TEP2, a "Best Current Practice" TinyOS Enhancement Proposal (TEP) document [79]. The application of the architecture to each hardware subsystem is codified in separate TEPs. They document the design considerations and mandatory specifications for the HIL layer interfaces for a particular hardware subsystem, and provide pointers to reference implementation of these interfaces in the TinyOS 2.x code base.

TEP	Title
TEP2	Hardware Abstraction Architecture
TEP101	Analog-to-Digital Converters (ADCs)
TEP102	Timers
TEP103	Permanent Data Storage (Flash)
TEP108	Resource Arbitration
TEP109	Sensors and Sensor Boards
TEP112	Microcontroller Power Management
TEP113	Serial Communication
TEP115	Power Management of Non-Virtualized Devices
TEP117	Low-Level I/O
TEP126	CC2420 Radio Stack
TEP131	Creating a New Platform for TinyOS 2.x

Table 4.1: TEPs specifying the hardware abstraction architecture and its implementation in TinyOS 2.x

Due to their platform-specific nature, the organization of the HAL layers is not subject to specification in TEPs. Despite this, the hardware subsystem TEPs often provide useful design suggestions and examples about the organization of the HAL components on particular platforms. Special focus is put on documenting useful hardware-independent interfaces for HAL and HPL components that enable writing platform-independent utility components, that can reduce the porting overhead. A typical example is the Timers TEP, which, in addition to the mandatory HIL specification, also documents a utility library for easy development of HIL and HAL abstractions on different hardware. Table 4.1 provides an overview of the current TEPs that relate to the specification and application of the hardware abstraction architecture.

Level of Portability

According to the portability anchor specification, the HIL interfaces should provide *full* hardware independence. The specification, however, defines the required API only at source code level and can effectively only mandate source code portability, like the POSIX interfaces.

In real-life use of the HIL abstractions, the expected *behavior* of the portable code written on top of the HIL abstractions plays equally important role as the code portability. To differentiate between those HIL abstractions that guarantee both source code and behavioral portability and the ones that only guarantee source code portability, the TinyOS 2.x community uses the concepts of "Strong HIL" and "Weak HIL" with the following meaning:

Strong HILs indicates that "portable code using these abstractions can reasonably be expected to behave the same on all implementations", which matches the original definition of the HIL level according to DASA. Examples include the HIL for the timer (*TimerMilliC*, TEP102), forLEDs (*LedsC*), active messages (*ActiveMessageC*, TEP116), sensor wrappers (*DemoSensorC*, TEP109), storage (TEP103), etc. Strong HILs may use platform-defined types if they also encapsulate their modification (i.e., they are platform-defined abstract data types), for example, the TinyOS 2.x message buffer abstraction, *message_t* (TEP111).

Weak HILs indicates that "portable code using these abstractions might exhibit platform-specific behavior". For example, the existing ADC abstraction requires platform-specific configuration and the returned data must be interpreted in light of this configuration. The ADC configuration is exposed on all platforms through the *AdcConfigure* interface that takes a platform-defined type, *adc_config_t*, as a parameter. However, the returned ADC data may be processed in a platform-independent way, for example, by calculating the max/min or mean of multiple ADC readings. So despite this platform-specific behavior, weak HILs still enable writing portable utility code, e.g., a repeated sampling for an ADC on top of the normal single-sample interface.

Code Organization

To provide the necessary context for the examples and the evaluation in the rest of the chapter, we briefly review the organization of the TinyOS 2.x code-base (Figure 4.5) and the association between its directory structure and the vertical and horizontal decoupling principles of the portability anchor.

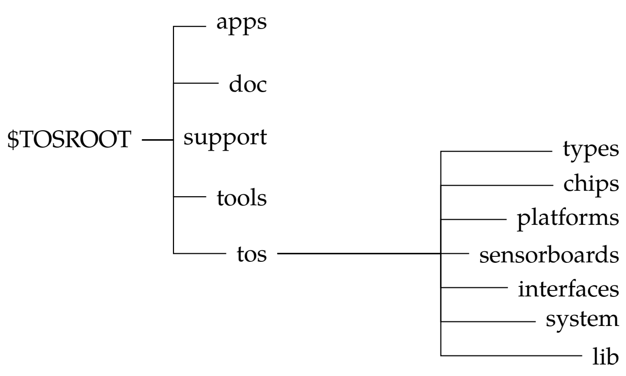


Figure 4.5: Directory structure of the TinyOS 2.x code-base.

- `/apps` contains a collection of standard platform-independent and platform-specific applications used for demonstration and release testing.
- `/tos` contains the core components of the OS including the ones for hardware abstraction, organized as follows:
  - `/tos/chips` contains the platform-independent, chip-specific abstractions of all supported hardware chips in the OS, following the horizontal decomposition. Each of these chip-specific abstraction, in turn, is comprised of a collection of components resulting from the three-level decomposition of the abstraction functionality, following the vertical decomposition principle.
  - `/tos/platforms` contains the platform-specific, chip-independent components for all supported hardware platforms, following the horizontal decomposition. Each `/tos/platforms` subdirectory contains a special ".platform" definition file that specifies the list of platform-independent chip abstractions and platform-specific "glue" components that comprise the platform.

- `/tos/interfaces` contains the system OS interfaces and the platform-independent HIL interfaces for all hardware subsystems, in effect specifying the top-level signature of the portability anchor as implemented in TinyOS 2.x.
- `/tos/system` contains basic system components like the scheduler and platform-independent HIL wrapper components.
- `/tos/lib` contains component libraries for different aspects of the system, including reusable utility code for building HIL and HAL abstractions, like the already mentioned timer library, in `/tos/lib/timer`, or the power manager library, in `/tos/lib/power`.

4.5.3 Vertical Decomposition Example

To illustrate the application of the vertical decomposition principles presented in Section 4.2 in the TinyOS 2.x code-base, we present the implementation of the platform-independent timer service for the Texas Instruments MSP430 MCU.

According to TEP102 [184], the portable HIL abstraction for timers in TinyOS 2.x is a millisecond-precision timer with a width of 32-bits. The precision is expressed in binary units, i.e. the millisecond HIL timer ticks 1024 times per second, and not 1000 times per second.

Due to the large diversity in the number of hardware timers on the WSN platforms and their features, the implementation of the HIL timer service is typically based on top of a single hardware timer which is then *virtualized* into as many timers as needed.

Figure 4.6 shows a simplified version of the component graph used to implement this service on the Texas Instruments MSP430 MCU. The graph concentrates only on the main components, their interfacing and classification in the three abstraction layers. Apart from *TimerMilliC* and *LocalTimerMilliC* which are portable and reside in `/tos/system`, all other components are chip-specific and reside in `/tos/chip/msp430/timer`. Many of them, however, internally use portable utility components from the timer library in `/tos/lib/timer`.

Hardware Timers

Below the hardware/software boundary, the MSP430 MCU offers two 16-bit hardware timers (Figure 2.2): *TimerA* with three Capture/Compare Registers (CCRs), and a *TimerB* with seven CCRs. Each timer can be driven by a separate clock source which can be configured with a limited set of prescalers. In TinyOS 2.x, the TimerA is typically connected to the same clock source as the CPU, but scaled down to 1 MHz, while the TimerB is typically driven by an external 32768 Hz quartz crystal. The HIL timer service on the MSP430-based TinyOS 2.x platforms is built on top of the TimerB using one of the seven available CCRs.

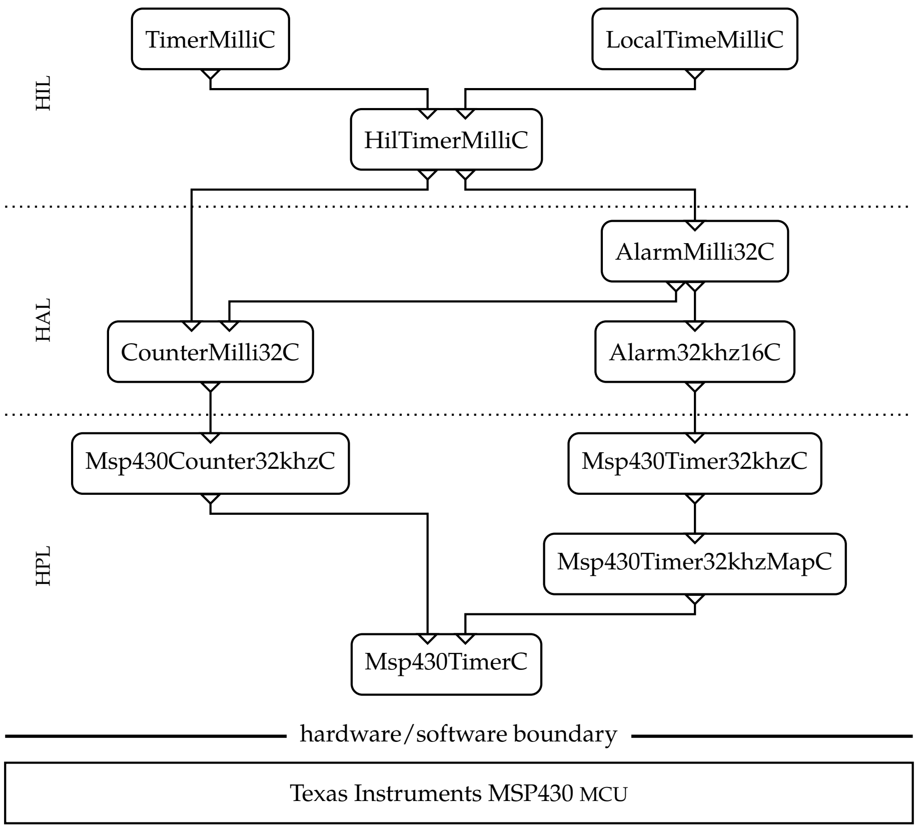


Figure 4.6: Vertical decomposition of the timer abstraction for the Texas Instruments MSP430 MCU.

HPL Components

Following the guidelines for HPL components (Section 4.2.1), the implementation of the timer abstraction starts by wrapping the low-level access to TimerA and TimerB in the form of a *Msp430TimerC* component which provides convenient configuration and service interfaces for the timers and their CCRs.

The *Msp430Timer32khzMapC* and *Msp430Timer32khzC* components reserve one of the TimerB CCRs from Msp430TimerC and export its services to the HAL level components. At the same time, *Msp430Counter32khzC* transforms the *Msp430Timer* interface for TimerB into a standard *Counter* interface (Figure 4.7). Counters count time with some precision and width, signaling overflow events when they happen.

As suggested by its name, the Counter provided by Msp430Counter32khzC has 32 kHz precision and performs no transformation on the width, maintaining the 16-bit size of the underlying hardware timer, which results in the signature *Counter<T32khz, uint16_t>*.

```
interface Counter<precision_tag, size_type>
{
  async command size_type get();
  async command bool isOverflowPending();
  async command void clearOverflow();
  async event void overflow();
}
```

Figure 4.7: The Counter interface: get() *command returns the current time,* isOverflowPending() *and* clearOverflow() *are used to check and clear a pending counter overflow flag, while the event* overflow() *signals that an counter overflow has happened in the current time.*

HAL Components

The low-level interfaces exported from the HPL components are subsequently used by the HAL component to build-higher level services. First, *CounterMilli32C* transforms the *Counter<T32khz, uint16_t>* service from Msp430Counter32khzC into a millisecond-precision, 32-bit wide Counter with the signature *Counter<TMilli, uint32_t>*.

At the same time, *Alarm32khz16C* creates a 32 kHz *Alarm* with a 16-bit size on top of the CCR exported by the HPL component Msp430Timer32khzC. Alarm components are extensions of Counters that signal an event when their CCR detects that the alarm time has been reached, forming the basis for precise, HAL-level timing (Figure 4.8).

```
interface Alarm<precision_tag, size_type>
{
  // basic interface
  async command void start(size_type dt);
  async command void stop();
  async event void fired();

  // extended interface
  async command bool isRunning();
  async command void startAt(size_type t0, size_type dt);
  async command size_type getNow();
  async command size_type getAlarm();
}
```

Figure 4.8: The Alarm interface: start(dt) *sets the alarm to fire in* dt *time units from the time of the command invocation,* stop() *stops any running timer and* startAt(t0, dt) *sets an alarm to fire in* dt *time units measured from some time* t0 *in the past. The event* fired() *signals that the alarm has expired.*

The Alarm provided by Alarm32khz16C, has the signature *Alarm<T32khz, uint16_t>*. The next HAL component, *AlarmMilli32C*, uses the services of CounterMilli32C to transform this Alarm to a new millisecond-precision, 32-bit wide Alarm. The resulting Alarm, with signature *Alarm<TMilli, uint32_t>*, represents the top public HAL interface.

HIL Components

As core HIL level component, *HilTimerMilliC* is responsible for the final transformation of the hardware-specific HAL service into a portable timer abstraction. As a first step, HilTimerMilliC transforms the alarm provided by the HAL into a *Timer*, the basic HIL-level timing service in TinyOS (Figure 4.9).

```
interface Timer<precision_tag>
{
  // basic interface
  command void startPeriodic(uint32_t dt);
  command void startOneShot(uint32_t dt);
  command void stop();
  event void fired();

  // extended interface
  command bool isRunning();
  command bool isOneShot();
  command void startPeriodicAt(uint32_t t0, uint32_t dt);
  command void startOneShotAt(uint32_t t0, uint32_t dt);
  command uint32_t getNow();
  command uint32_t gett0();
  command uint32_t getdt();
}
```

Figure 4.9: The Timer interface: startOneShot(dt) *and* StartPeriodic(dt) *start a single or periodic timer with duration* dt *units from the time of invocation; their extended versions* startOneShotAt(t0,dt) *and* StartPeriodicAt(t0,dt) *start timers anchored at some instant* t0 *in the past. Similarly to the Alarm interface, the event* fired() *signals that the timer has expired for the "OneShot" timers, or that it was repeated for the "Periodic" timers.*

In the second step of the implementation, HilTimerMilliC virtualizes the single *Timer<TMilli>* into 255 virtual timers and offers them through a parametrized interface *TimerMilli[uint8_t]*. For this, it stores the deadlines for all of the timers it provides and schedules the underlying single timer to fire at the next upcoming deadline. In addition, the HilTimerMilliC offers a *LocalTime<TMilli>* interface that provides a service that is similar to a 32-bit Counter (without the *overflow()* event), and is typically used to track the time expired from the last system boot.

Finally, *TimerMilliC* and *LocalTimeMilliC*, are platform-independent wrappers that simplify the client wiring to the interfaces exported by HilTimerMilliC and provide the top hardware-independent interface for the timer abstraction.

Abstraction Overhead

By following the vertical decomposition principles, the resulting component stack provides both convenient and portable millisecond timer service through the virtualized Timer abstraction at HIL level, as well as streamlined access to the platform's hardware timers through the Counter and Alarm concept at HAL.

An important difference between the Alarm and Counter HAL service and the HIL Timer service is the concurrency class of their commands and events. To guarantee optimal control over the underlying hardware timers, all commands and events from the HAL interfaces are *asynchronous*, i.e. they execute in interrupt context. In contrast, the commands and the events of the HIL Timer interface are executed in *synchronous* context, as part of *tasks*, the deferred execution mechanism provided by nesC. As a result, their execution latency is more sensitive to other code in the system because tasks run to completion and can not be interrupted by other tasks.

In Section 4.6.4 we illustrate the practical effects from the loss of control fidelity between the HIL and HAL interfaces using a simple micro-benchmark application for Pulse-Width Modulation (PWM) servo motor control.

4.5.4 Horizontal Decomposition Example

To exemplify the application of the horizontal decomposition principles presented in Section 4.3 and their impact on the porting process, we briefly summarize the integration of the *cc2420* chip—the platform-independent abstraction for the CC2420 radio transceiver—on the *telosb* platform. The CC2420 radio transceiver has multiple physical interfaces and is thus representative of the binding requirements for similarly complex chips.

Chips

Following the horizontal decomposition rules, the physical interfaces of the CC2420 transceiver are reflected as external services on which the cc2420 chip depends. The existing cc2420 implementation requires seven such services. Some of them, like Random Number Generator (RNG), Finite-State Machine (FSM) and Light Emitting Diode (LED) are provided by platform-independent components in `/tos/system`. The remaining ones, however, have to be wired by the platform. In particular, the cc2420 chip requires platform binding for:

- the SPI bus used for accessing the transceiver's packet buffers and command registers;
- the GPIO pins used for signaling time-critical events like Clear Channel Assessment (CCA), Start Frame Delimiter (SFD), First In, First-Out (FIFO) buffer overflow, etc.;
- the 32 kHz Alarm used for timing Carrier Sense Multiple Access (CSMA) back-off and acknowledgments timeouts; and
- the interrupts that are triggered by the chip.

On the telosb platform, these services are provided by the MSP430 MCU, which is abstracted in the form of a platform-independent *msp430* chip. Therefore, the role of the platform-specific telosb glue code becomes one of binding the services on which the cc2420 chip depends to the ones provided by the msp430 chip.

Platform

Figure 4.10 illustrates the implementation of this service binding. The telosb platform provides four glue components that bridge between the provided and the used interfaces on the two chips:

- *HplCC2420SpiC* wires the *CC2420SpiWireC* component in `/tos/chips/cc2420/spi` to the platform-independent abstraction of the SPI bus, provided by *Msp430Spi0C* in `/tos/chips/msp430/usart`;
- *HplCC2420AlarmC* wires the *AlarmMultiplexC* component in `/tos/chips/cc2420/alarm` to the 32 kHz-precision, 32-bit wide Alarm service provided by *Alarm32khz32C* in `/tos/chips/msp430/timer`;
- *HplCC2420PinsC* wires the GPIO pins used in various cc2420 components like *CC2420ContolC*, *CC2420ReceiveC*, *CC2420TransmitC*, etc., to the msp430 GPIO pins abstracted as *Msp430GPIOC* components in `/tos/chips/msp430/pins`; while
- *HplCC2420InterruptsC* wires the interrupts generated by the above components to msp430 interrupt-enabled pins, abstracted as *Msp430InterruptC* components in `/tos/chips/msp430/pins`, as well as to a CCR register for SFD timing, abstracted as *GpioCaptureC* in `/tos/chips/msp430/timer`.

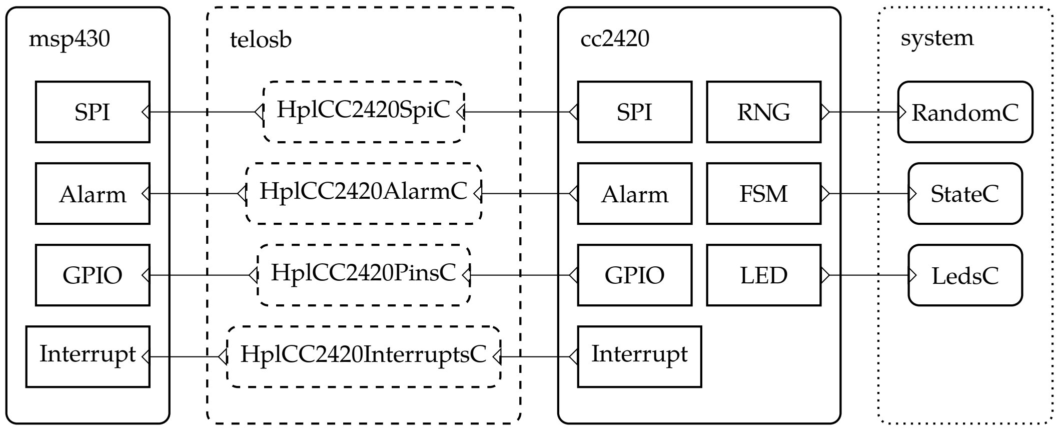


Figure 4.10: Platform-specific components for binding the cc2420 and msp430 chip abstractions on the telosb platform.

These platform-specific glue components, actually reside in the telosa platform directory, in `/tos/platform/telosa/cc2420`, which the telosb platform inherits.

Code Reuse

The amount of binding code that is required to “attach” a complex chip like the cc2420 to a given MCU can give an approximate indication for the overall effort that a developer needs to invest in order to build a new platform out of already available chip abstractions. As long as this non-portable glue code remains small compared to the portable code written for each chip, the horizontal decomposition results in significant reuse gain which reduces the porting overhead for new platforms. For the above example, the four telosb glue components with 74 Source Lines of Code (SLOC) are only 1.6 % of the total size of the cc2420 chip, abstraction which has 4704 SLOC. Attaching the same chip on the micaz platform requires only 154 SLOC in glue components, which is only 3.3 %. In Section 4.6.3 we provide a more detailed evaluation of the ratio between chip-specific and platform-specific code for several hardware modules and platforms.

4.6 Evaluation

A realistic evaluation of the DASA portability anchor, like any other architectural proposal, is a challenging task that ultimately requires collection and analysis of feedback from developers with long exposure to the design, gathered through development and use of multiple independent implementations of the architecture.

Lacking the necessary time perspective for a comprehensive “look-back” study, the evaluation presented in this section leverages the TinyOS 2.x code-base, as a mature and widely-used implementation of the proposed architecture by large pool of developers, to asses how well the DASA portability anchor achieves the main design objectives of:

- enabling effective abstraction of the hardware resources on the underlying platforms;
- supporting easy development of portable applications that can run with minimal changes on different platforms; and
- offering flexible control over the abstraction/performance trade-offs in the lowest levels of the software stack.

4.6.1 Composing Portable Applications

To illustrate its core features and demonstrate the level of achievement of the first two objectives, we have performed a comparative analysis of the portability anchor’s implementation in TinyOS 2.1.1, focusing on the static resource utilization in the different component categories resulting from the application of the vertical and horizontal decomposition rules, for several portable applications and hardware platforms.

In the following, we first briefly summarize the used evaluation methodology and metrics, before presenting the results of the study, organized in two views reflecting the two main decomposition principles.

Test Applications

A substantiated assessment of the design goals achievement requires evaluation of the portability anchor's implementation using a broad range of test applications, representative of the service requirements that typical WSN applications have from the underlying hardware platform.

The set of testing applications needs to be carefully selected to maximize the informative value of the evaluation results. First of all, the applications should "exercise" the most important hardware subsystems, providing opportunity to assess how successfully the architectural principles of the portability anchor can be applied for abstracting the services of different hardware resources. In this, the testing applications have to strike good balance between simplicity and realism. The first one allows easier observation of the effects that the architectural decisions have on the resulting implementation, while the second one provides better insight into the involved challenges. Finally, the applications should have a well defined behavior and sufficient visibility, facilitating easy replication and comparison with alternative solutions.

To satisfy these requirements, we have based our evaluation on a set of six standard applications distributed with the TinyOS 2.x code-base. These applications, like their TinyOS 1.x predecessors, have a tradition of being used as a de facto benchmarking suite in various areas of WSN research [23, 41, 77, 92, 139, 140, 186, 215, 220]. Below, we provide a short description for each of the used applications, focusing on the leveraged hardware subsystems and the portability anchor features they help illustrate.

Null This is an empty application skeleton with no application-level logic. The application tests the initialization sequence and demonstrates the proper functioning of the automatic power management of the MCU and the peripheral hardware devices (Section 4.4). The Null application component is not leveraging any hardware services. Thus, the results establish a useful baseline for evaluating the resource usage in the different hardware systems used by the remaining test applications.

Blink A simple application that blinks three platform LEDs on the overflow events from three independent timers with periods of 1000 ms, 500 ms and 250 ms. It tests the proper functioning of the scheduler and allows us to evaluate the abstraction of the timer system (Section 4.5.3) and the LEDs.

RadioCountToLeds The application can be used for coarse assessment of the radio connectivity between two nodes. On the sender side, it increments and broadcasts a counter value every 250 ms. On the receiver side, it displays the three low-order bits of the received counter on the platform LEDs. In our study, RadioCountToLeds facilitates the evaluation of the transceiver abstraction(Section 2.1.1), in addition to the timer system.

Oscilloscope This is a basic data-collection application that periodically samples the platform's default sensor every 250 ms. The sensor readings are buffered and after 10 readings they are broadcasts as a message over the transceiver. Oscilloscope enables us to evaluate the abstractions for the ADC and the sensing elements (Section 2.1.1), in addition to the abstractions for the timer system and the transceiver.

StorageLog The application performs a set of random log record writes to the external storage using the StorageLog abstraction (TEP103). Subsequently, it reads the log and tests the records for correctness. The results from the test are conveyed through the platform LEDs and by sending a status message over the serial interface. In our study, the StorageLog is used to asses the abstraction of the external storage element (Section 2.1.1) and the serial stack (Section 2.1.1).

BaseStation transfers packets between the serial and the transceiver interfaces, at the platform-independent Active Messages abstraction (TEP116) level. The application includes queues in both directions that enable more graceful handling of the traffic load spikes. BaseStation allows us to evaluate the hardware abstraction code for the transceiver and the serial interfaces.

Table 4.2 provides a summary of the hardware resources used by the different test applications. It shows that the selected application set provides sufficient coverage of the hardware abstraction code and allows comparative analysis across different hardware resources.

Test Application	Hardware Resources
Null	
Blink	timer, LEDs
RadioCountToLeds	transceiver, timer, LED
Oscilloscope	sensor, transceiver, timer, LED
StorageLog	storage, serial, timer, LED
BaseStation	serial, transceiver, timer, LED

Table 4.2: Overview of the main hardware resources used in each of the test application.

Test Platforms

Similarly to the test applications, the selection of the testing platforms requires careful balancing between two conflicting goals. From one side, the platforms need sufficient variability in the hardware elements, to demonstrate the portability anchor's capacity for abstracting differences in diverse platforms behind a common hardware-independent interface. From the other side, the test platforms must have enough common hardware elements to illustrate the reuse gains in the porting process enabled by the horizontal decomposition of the portability anchor.

To satisfy these goals, we have selected a set of five prominent WSN hardware platforms: mica2, micaz, telosb, eyesIFXv2.1 and intelmote2, as our evaluation targets. They provide a representative sample of the WSN design space, as confirmed by our platform survey (Section 2.2). Table 4.3 briefly summarizes the hardware chips used for the processing element, the transceiver and storage on each platform and highlights the commonalities. The test platforms have been reviewed in greater detail in Section 2.1.2.

	mica2	**micaz**	**telosb**	**eyesIFXv2.1**	**intelmote2**
Processor	*ATmega128L*	*ATmega128L*	*MSP430F1611*	*MSP430F1611*	*PXA271*
Transceiver	*CC1000*	*CC2420*	*CC2420*	*TDA5250*	*CC2420*
Storage	*AT45DB041B*	*AT45DB041B*	*STM25P80*	*AT45DB041B*	*PXA27XP30*

Table 4.3: Common components on the mica2, micaz, telosb, eyesIFXv2.1 and intelmote2 designs.

As shown in Table 4.3, the five test platforms feature three different processing architectures: mica2 and micaz share an 8-bit Atmel Atmega128L MCU, telosb and eyesIFXv2.1 both use an 16-bit Texas Instruments MSP430F1611, while the intelmote2 is based on a much more capable, 32-bit Intel PXA271 CPU.

Three of the five platforms—micaz, telosb and intelmote2—share the same Chipcon CC2420 transceiver, while mica2 and eyesIFXv2.1 have narrow-band transceivers. The mica2's Chipcon CC1000 is byte-oriented and interfaces through the SPI bus for both control and data. The Infineon TDA5250—used on the eyesIFXv2.1—is bit-oriented, and is interfaced through the UART module on the MCU which serves as hardware accelerator.

The storage elements on the test platforms show comparable level of diversity. The Atmel AT45DB041B serial flash is reused on three platforms: mica2, micaz and eyesIFXv2.1. The telosb uses the ST Microelectronics STM25P80, while intelmote2 leverages an internal PXA27XP30 flash chip as storage.

In summary, the selected platforms offer a good trade-off between design diversity and component reuse, allowing us to illustrate the challenges and benefits associated with the implementation of the two decomposition principles of the portability anchor.

Component Classification

The first step in the evaluation process involves the extraction of the employed software components and their classification into categories, according to the decomposition principles of the portability anchor. We used the nesC compiler's dump option `-fnesc-dump`, to extract the component graphs for each test application on each hardware platform, and leveraged this data to create a master list of 376 unique non-generic components used in the implementation. These components were subsequently classified in different categories (Table 4.4).

Decomposition	Component Categories
Vertical	HPL, HAL, HIL, application, system
Horizontal	chip, platform, application, system

Table 4.4: Classification of the components in different categories according to the vertical and horizontal decomposition principles of the portability anchor.

The "HPL", "HAL" and "HIL" tags denote that the component belongs to the respective layer in the vertical decomposition of the portability anchor. The "application" tag denotes application-level components, while the "system" category is reserved for system components like the ones implementing the scheduler or other services not dependent on the underlying hardware resources.

Correspondingly, for the horizontal decomposition, the "chip" tag denotes that the component belongs to a platform-independent, chip-specific abstraction of a hardware resource. The platform-specific configuration and binding components are marked with the "platform" tag. The "application" and the "system" tags have the same semantics as in the vertical classification.

The classification of the non-generic components in this master list was performed in a semi-automatic fashion. Although TEP3 [221] includes useful naming guidelines—like prefixing the Hardware Presentation Layer components with the HPL acronym, and the Hardware Adaptation Layer components with the chip name—these rules are not consistently followed in the TinyOS 2.x code-base. As a result, the identification of the vertical decomposition level of a given component is not straightforward and can not be easily automated. Thus, we have performed this step of the classification by manual annotation. The classification of the components according to the horizontal decomposition rules can be performed much easier, because the category is readily deduced from the component location in the directory tree (Section 4.5.2).

In contrast to the non-generic components, the proper classification of the generic components—frequently used in the hardware abstraction code—is more challenging. The implementation of the test applications, for all test platforms, employs 51 unique generic components. Like classes in object-oriented systems, the nesC generic components can be instantiated multiple times and the category of the instance can vary. Hence, both the classification and the calculation of the evaluation metrics has to be performed on an instance-by-instance basis. Using the instantiation tree information extracted from the nesC dump files, and for each unclassified generic component instance, we have backtraced over the instantiation branch—potentially across several nested instantiation levels—up to a parent component in the master component list with a known category. Subsequently, all generic instances in the chain were annotated with the category of this parent component.

Metrics

To asses the implementation's conformance with the architectural specification and the level of achievement of the core design goals, we have evaluated three metrics reflecting the static resource usage in the different vertical and horizontal component categories for each application/platform combination. In the following we overview their definitions and the methodology we used for their calculation.

Source Lines of Code We use the physical Source Lines of Code (SLOC) metric as an approximate indication for the developer effort that needs to be invested for the implementation of a given component [5]. Our SLOC evaluation is concentrated on the nesC code and does not take into account external header or source files written in C. For the counting of the SLOC in a given nesC component, we leveraged the *SLOCcount* tool [213], using the same source line counting logic as for C source files.

Code size The SLOC metric is sensitive to coding style preferences and can exhibit high variability even for similar algorithmic content. Many of these differences, however, are reduced during the compilation and linking process. The elimination of dead code and other optimization steps also contribute to leveling out the diversity in the source code. Thus, the code footprint of the resulting binary can be a better indication for the static resource usage in the different component categories and correlates better with more advanced code complexity measures [131].

The compilation and optimization steps in nesC, unfortunately, also destroy the component boundaries, making the attribution of binary code to individual source-level components very challenging. To reestablish the link, we disabled the nesC inlining and analyzed the names and sizes of the symbols in the code sections of the binaries, as reported by the *nm* tool from the GNU binutils suite [59].

Data size Due to the static allocation nature of TinyOS, the size of the data symbols in each component is a good indication of the total "state" required for implementing the component logic. To calculate the RAM footprint of each component, we applied the same approach as for the code size metric, only focusing on the names and sizes of the symbols in the data segment.

Since the SLOC metric is not context-dependent, it was independently calculated for each entry in the list of unique components that we leveraged in the classification step. The code size and data size metrics were evaluated individually for each application/platform combination. This information was then fused with the SLOC counts and the category tags to produce a summary statistics for the three metrics per vertical and horizontal decomposition category.

We analyze the obtained results in the next two sections, organized along the two decomposition axes.

4.6.2 Vertical Decomposition View

This section presents the static resource usage in the different vertical component layers of the portability anchor's architecture. The data is faceted in a grid structure, enabling easy visual comparison of the metrics across the different test applications (for a specific test platform) and across the different test platforms (for a specific test application).

Total Development Effort

The results from the evaluation of the SLOC counts in the different component layers according to the vertical decomposition of the portability later are shown in Figure 4.11.

The total development effort for the selected test applications on the target platforms ranges from relatively low to moderately high. At the one end, the implementation of the baseline Null application on the intelmote2 platform requires only 1674 SLOC. At the other end, the components used in the implementation of the BaseStation application on the eyesIFXv2 have sizeable 13938 SLOC.

For all applications and platforms, the bulk of this effort is concentrated in the components comprising the portability anchor. Ignoring the Null baseline, the ratio of the SLOC in the portability anchor's components with respect to the total SLOC ranges from 85.2 % for Blink on intelmote2 to 97.1 % in the case of RadioCountToLeds on eyesIFXv2. The results confirm that these components encapsulate significant amounts of intricate hardware abstraction code behind their interfaces, which enables more efficient development of higher-level code. The ratio between the SLOC in the application components and the one in the portability anchor averages 1.9 % across the different target platforms. Although these numbers are also influenced by the complexity of the selected test applications, the evaluation of the DASA interoperability anchor's prototype (Section 5.6), confirms that even in the case of more complex portable applications, the hardware abstraction code remains a significant fraction of the total development effort.

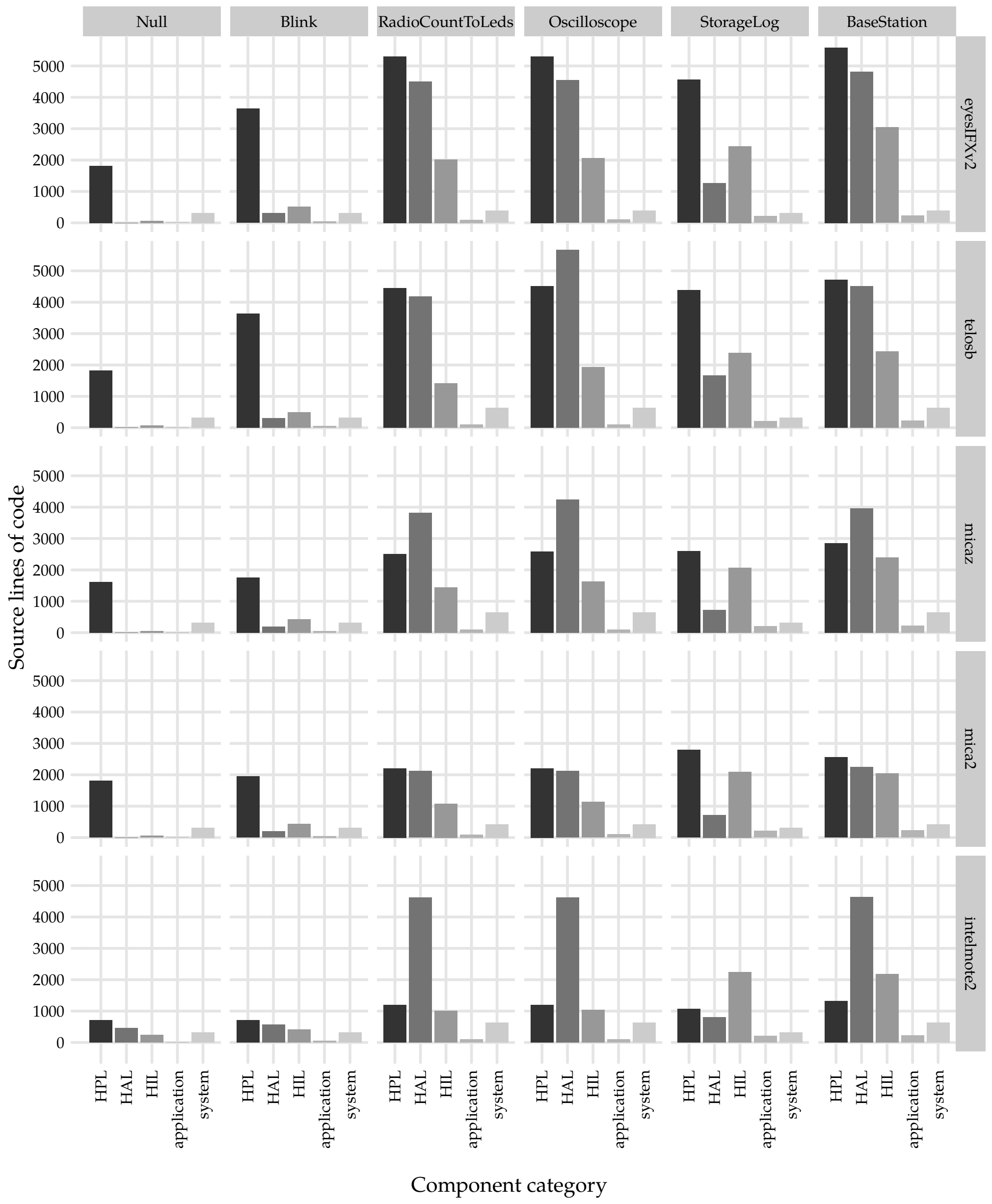


Figure 4.11: Vertical decomposition of six TinyOS 2.x applications on five hardware platforms: source lines of code.

HPL Overheads

In terms of the internal distribution of the development effort among the three component layers in the vertical decomposition, the evaluation results seem to indicate very high SLOC overheads for the HPL components. Following the design guidelines (Section 4.2) the bulk of the development effort is expected in the writing of HAL abstractions and HIL portability wrappers. The results, however, show high SLOC counts in the HPL components which contribute between 16.2 % (BaseStation on intelmote2) and 81.9 % (Blink on telosb) of the total portability anchor's code.

A closer inspection of the code shows that the elevated SLOC counts in the HPL components are mainly due to the selected methodology for evaluating generic component instances. The individual accounting of the SLOC in each instance (necessary for a more fair comparison with the remaining evaluation metrics) overstates the significance of generic HPL code that has low algorithmic complexity.

Platform specifics also play a significant role. For example, on the mica2 platform, the HPL component of the transceiver is included in the default platform initialization, independently from the fact if the application uses the transceiver or not, elevating the HPL counts. Similarly, the timer abstractions on the MSP430-based platforms carry high HPL overhead due to the flexible low-level wrapping of the individual hardware timers. In contrast, the ATMega128-based platforms use a single dedicated hardware timer for the creation of the platform-independent Timer service, leading to simpler HPL components (139 SLOC in comparison to 1858 SLOC for the MSP430 platforms).

HAL and HIL Overheads

The development effort for the HAL components mainly depends on the complexity of the hardware chip and on the abstraction level of the raw services provided by the hardware, which are wrapped by the HPL. The large jump in the SLOC counts of the HAL components between Blink and RadioCountToLeds confirms that the transceiver is typically the most complex hardware resource on the platform, and illustrates the large development effort required for encapsulating its services behind more convenient high-level abstractions. The HAL components make on average 47.2 % of the total portability anchor SLOC in RadioCountToLeds across the different platforms.

In comparison to the transceiver, the increase in HAL counts due to the ADC and sensor abstractions in Oscilloscope is almost negligible. Here, the contribution of the HAL components jumps only to 48.3 % of the total SLOC. The inclusion of the serial stack in BaseStation has similarly low impact on the HAL counts.

For the majority of the evaluated hardware subsystems, the invested effort in the development of the streamlined domain-specific abstractions in the HAL is larger than the effort for the development of the portability equalizers in the HIL. The ratio of the SLOC counts in these two component levels can be indicative of the alignment between the current HIL contract in TinyOS and the platform-specific HAL interfaces on each platform. For the evaluated applications, this ratio ranges between 1.03 for mica2 and 1.81 for intelmote2.

Blink and StorageLog are exceptions, with "inverted" HAL vs. HIL ratio of 0.77 and 0.43, respectively. The reasons lay in the relatively high level of abstraction directly provided by the raw hardware interfaces. In the case of the timer system, the HAL Alarm and Counter abstractions are very close to the services provided by the underlying hardware, so the components are relatively thin. In comparison, the vitalization of the HAL services necessary for creating the platform-independent HIL Timer service (Section 4.5.3) requires more effort, resulting in this inverted HIL to HAL ratio. Similarly, in the StorageLog case, the HAL level services towards the external storage are close to the low-level services exported at the HPL level, while significant effort is concentrated in the development of the portable Log abstraction on top of the HAL. The ratio is further amplified by the internal decomposition of the serial stack which follows a similar pattern.

Code Footprint

Figure 4.12 summarizes the evaluation results for the code size metric. As discussed in the metrics section, it provides better indication of the intrinsic algorithmic complexity in the components than the SLOC counts, but is sensitive to the CPU architecture and the used compiler and linker suite.

The disabling of the nesC cross-component inlining—necessary for establishing a link between the binary image and the source-level components—has varying impact on the code size for different toolchains. For the MSP430-based platforms using the msp430-gcc toolchain, the non-inlined binaries are on average 1.58 times larger than their inlined versions. On the ATmega128-based platforms, with the avr-gcc toolchain, the increase is much larger and averages 3.77 times. On the intelmote2, with the xscale-elf-gcc toolchain, the increase is 2.50 times.

Despite these influences, the relative ratio of the code sizes in the different component categories can be compared across applications on a single platform, and across different platforms, as long as they share the same CPU architecture and toolchain: i.e. across eyesIFXv2 and telosb; and across mica2 and micaz. The obtained results mainly corroborate the conclusions from the SLOC study. Across all applications and platforms, the code in the portability anchor's components contributes on average 84.8 % of the total binary size.

The internal distribution of the code size between the different component categories shows much lower contribution of the HPL components (on average, 27.6 % of the total code size) than for the SLOC metric (on average 51.7 % of the total source lines of code), thanks to the low algorithmic complexity and the dead-code elimination in the compilation step. The average ratio between the footprint of the HAL and the HIL components ranges between 0.74 for micaz and 1.84 for intelmote2.

Similarly to the SLOC metric, the transceiver remains the main contributor to the HAL and HIL footprints in RadioCountToLeds, Oscilloscope and BaseStation. The difference in the code sizes for these applications between micaz and mica2 illustrates the higher complexity of the CC2420 transceiver abstraction vs. the one for the CC1000. At the same time, the comparison between telosb and eyesIFXv2 shows consistently higher footprints for the TDA5250 transceiver that requires extensive software support for interfacing and hardware acceleration through the UART interface. On eyesIFXv2 and mica2, the CCA is performed in software by direct sampling

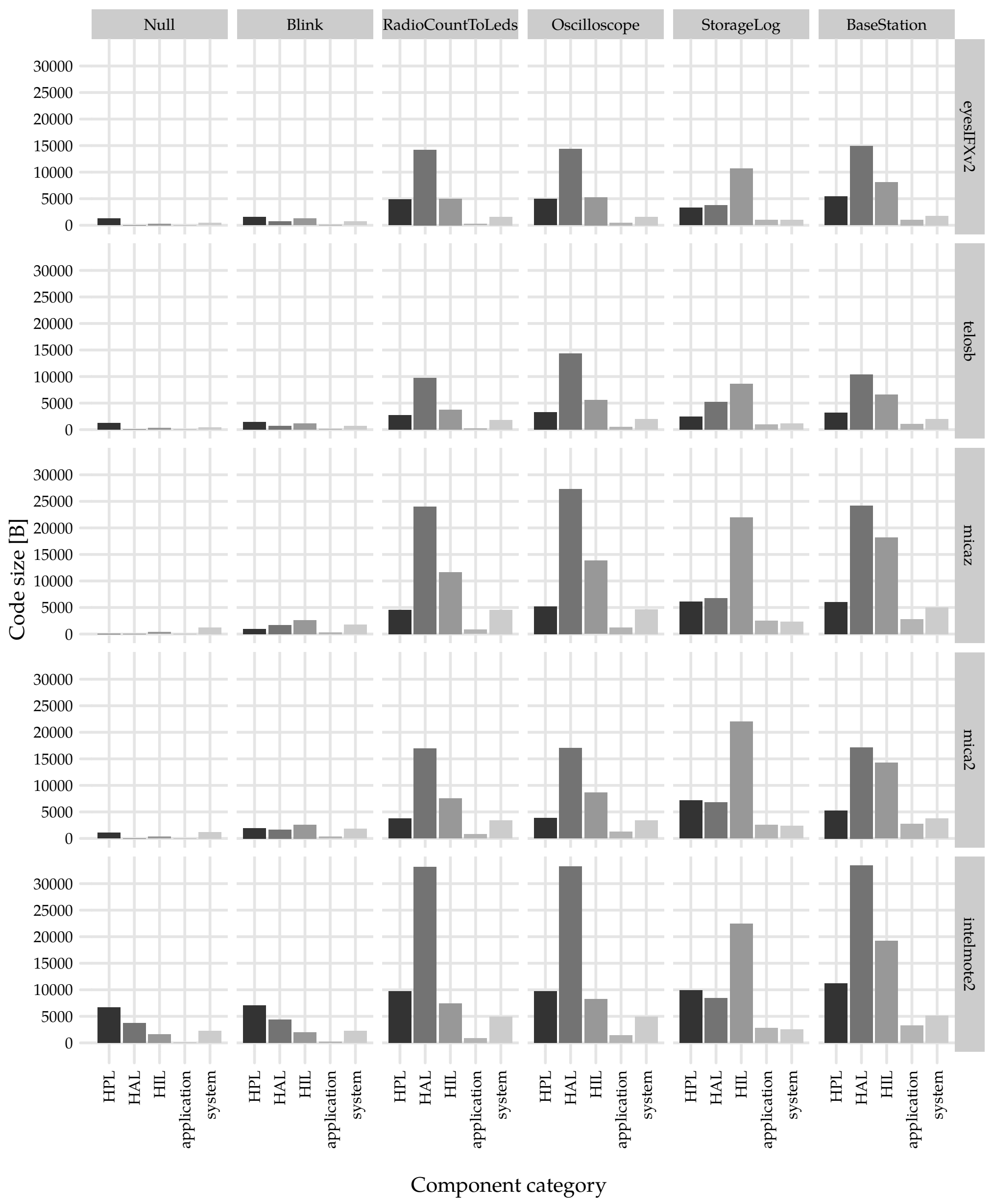


Figure 4.12: Vertical decomposition of six TinyOS 2.x applications on five hardware platforms: code size.

of the RSSI signal of the transceiver. As a result, Oscilloscope and RadioCountToLeds show almost no difference in the code footprint because the sensor stack is already incorporated in the transceiver code. The comparison of the results between Radio-CountToLeds and BaseStation on all platforms shows that the serial stack mainly contributes to the HIL footprint, due to the relatively low level of abstraction in the UART interfaces, combined with a more complex encoding, framing and dispatching framework for serial communication which is built on top.

Memory Footprint

The data size evaluation results are presented in Figure 4.13 and provide additional insight in the state requirements in the different component levels of the portability anchor. The results show that for the majority of test applications, the bulk of the data state is maintained by the portability anchor's components. Ignoring the Null baseline, the state allocated in these components ranges between 45 bytes for Blink on micaz and 753 bytes for BaseStation on eyesIFXv2. As expected, StorageLog and BaseStation exhibit disproportionately large state allocations at application level. Their application components use this memory for storing the test log records and message queues, respectively The memory footprints of their application components average 1101 bytes and 1387 bytes, across the different platforms.

The results confirm that with small exceptions, the implementation follows the design guidelines for keeping the HPL components stateless. With 15 B of space, the largest violation happens in the SPI interfacing component for the Atmel AT45DB041B storage chip. The distribution of the memory footprint between the HAL and the HIL components mirrors the results for the code size metric. Due to the transceiver abstraction, the bulk of the state allocated in RadioCountToLeds and Oscilloscope (on the average, 75.7 % and 77.9 % of the total allocation in the anchor, respectively) resides in the HAL components that implement the complexity-hiding abstractions and perform concurrency and power management. In contrast, the timer abstraction and the serial stack in Blink and StorageLog have larger footprints in the HIL portability wrappers (on the average 75.4 % and 82.9 % of the total allocation in the anchor, respectively), while BaseStation has almost balanced allocation in the two component categories.

Summary

The presented evaluation results from the vertical decomposition illustrate the positive impact from the increased rigidity in this part of the software architecture that is introduced by the portability anchor concept. They corroborate our claims for successful achievement of the main design goals by showing that:

- The hardware abstraction code in TinyOS successfully follows the vertical decomposition principles of the design: the low-level interaction with the hardware is wrapped in HPL components, on top of which streamlined complexity-hiding HAL abstractions are being built, and subsequently transformed into hardware-independent services, through the portability wrappers in the HIL;

- The implementation provides usable interfaces to the resources on the test platforms. The exported HAL abstractions match the hardware service requirements of WSN applications and simplify the development process by hiding the intricate low-level interaction with the hardware; and

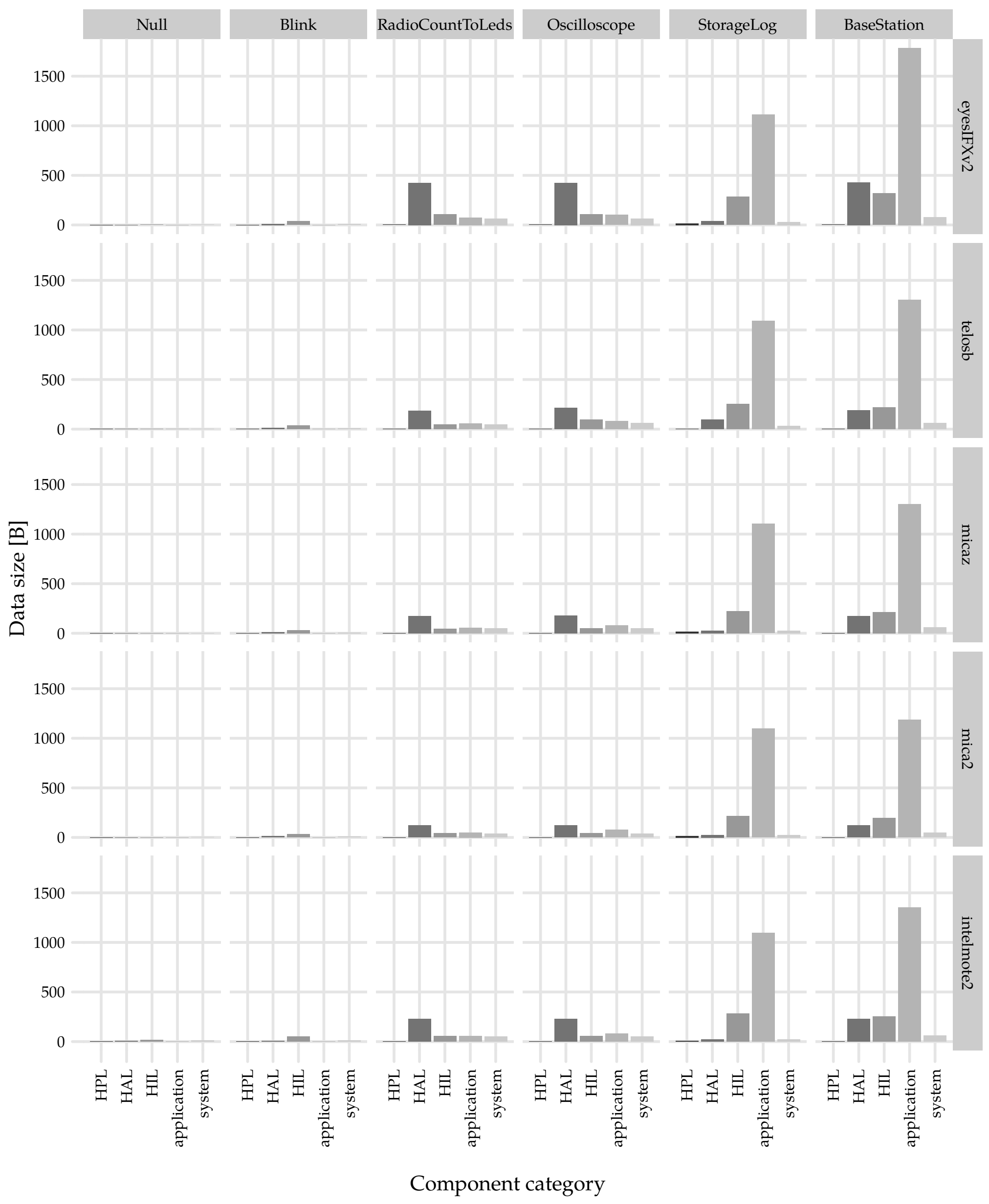


Figure 4.13: Vertical decomposition of six TinyOS 2.x applications on five hardware platforms: data size.

- The HIL wrappers effectively mask the diversity between the test platforms. Their hardware-independent interfaces decouple the software development from the evolution of the underlying hardware and pave a way for developing portable system services and applications.

4.6.3 Horizontal Decomposition View

The main objective of the portability anchor's horizontal decomposition (Section 4.3) is the establishment of an architectural basis for a more efficient platform porting process. In the following, we presents the results from our analysis of the static resource usage in the different component categories resulting from this decomposition, and argument how they support our claims for successful achievement of the core design objective.

Development Effort

Figure 4.14 summarizes the results from the SLOC count evaluation. As already discussed in Section 4.5.4, in this context, we use the SLOC metric as approximate indication for the amount of effort the platform developer has to invest in order to build a new platform out of already available chip abstractions. A low number of SLOC for the platform-specific inter-chip binding (development effort invested once per platform) is desirable, when compared with the connectivity requirements of the involved platform-independent chips and the number of SLOC in their abstractions (development effort invested once for many platforms).

Across all test applications and platforms, the results clearly show that the development effort is strongly concentrated in the platform-independent chip abstractions. Excluding the Null baseline, their contribution in the total SLOC ranges from 52.8 % for StorageLog on intelmote2 and 82.8 % for Blink on eyesIFXv2. Correspondingly, the portability of the code is very high. The ratio of the SLOC counts in the platform components—from one side—and the platform-independent chip and system components—from the other side—ranges between 2.51 % for Oscilloscope on telosb and 25.0 % for Blink on intelmote2.

The comparison of the results for RadioCountToLeds with the ones for Oscilloscope and BaseStation confirms that the transceiver is the most complex hardware resource and responsible for the bulk of the chip SLOC counts. Only on mica2, the more simple CC1000 chip abstraction has a comparable number of SLOC with the one for the external storage and serial stack as represented by StorageLog.

Across platforms, the transceiver is also the most demanding chip in terms of the interconnection requirements. As a result, the binding and configuration components for the transceiver form the bulk of the SLOC counts in the platform components in RadioCountToLeds, Oscilloscope and BaseStation. The binding requirements, however, vary between the different transceiver chips. For example, due to the low-level of abstraction, the TDA5250 on eyesIFXv2 requires more extensive interconnect with the rest of the platform than the more compact CC2420 and CC1000 designs on the other platforms. Consequently, the eyesIFXv2 features the largest absolute platform-specific SLOC count in these three applications.

When compared with the results for StorageLog, it is evident that the binding and configuration requirements for the external storage and the serial interfaces are more modest than the ones for the transceiver, across the different platforms.

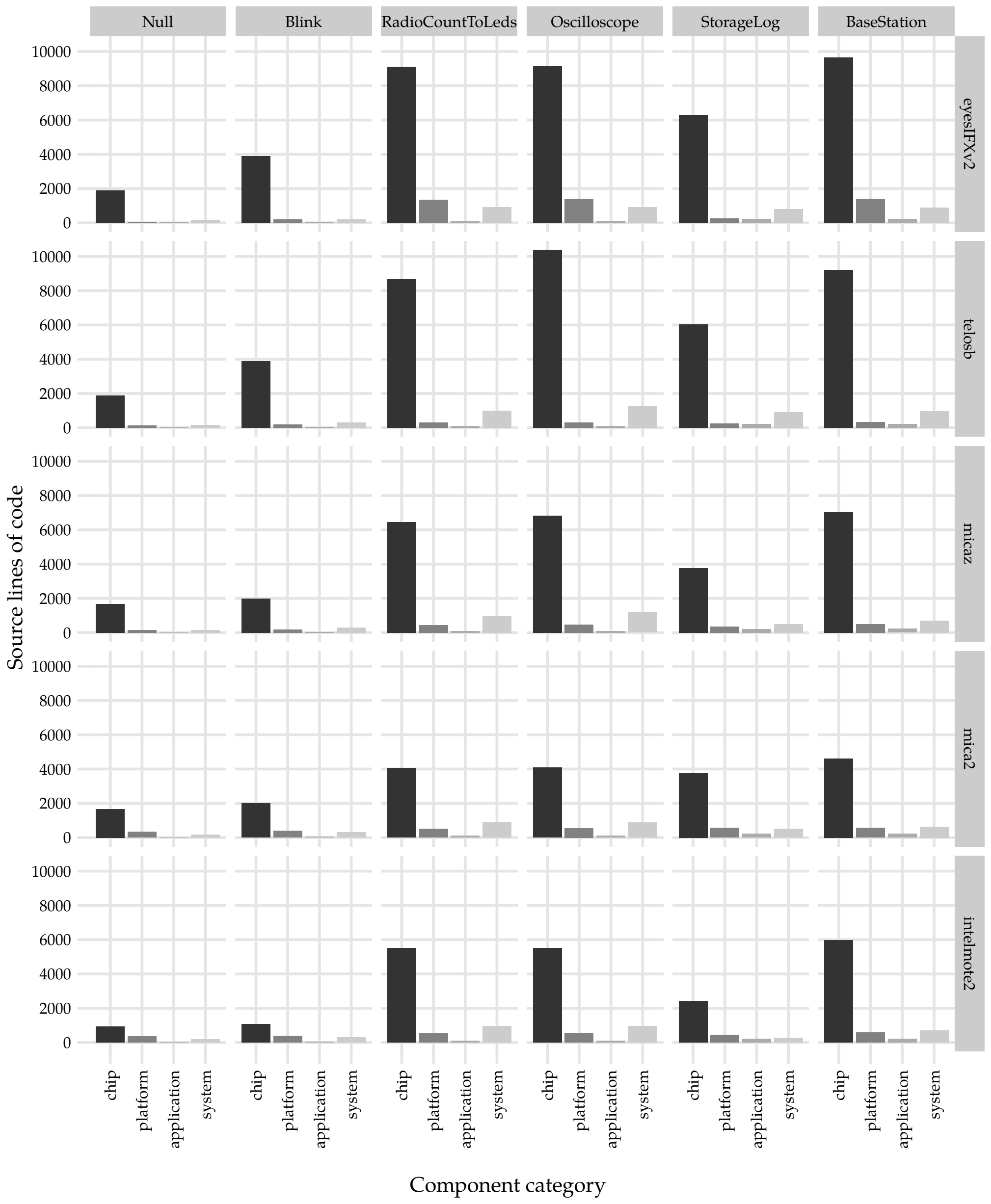


Figure 4.14: Horizontal decomposition of six TinyOS 2.x applications on five hardware platforms: source lines of code.

Code Footprint

The results from the code size evaluation are presented in Figure 4.15 and strengthen the above insights. For all applications, on all platforms, the amount of platform-independent code (chip, application and system) outweigh by a large margin the platform-specific code. The chip abstractions contribute between 40.8 % (for Blink on mica2) and 75.1 % (for Oscilloscope on telosb) of the total code size.

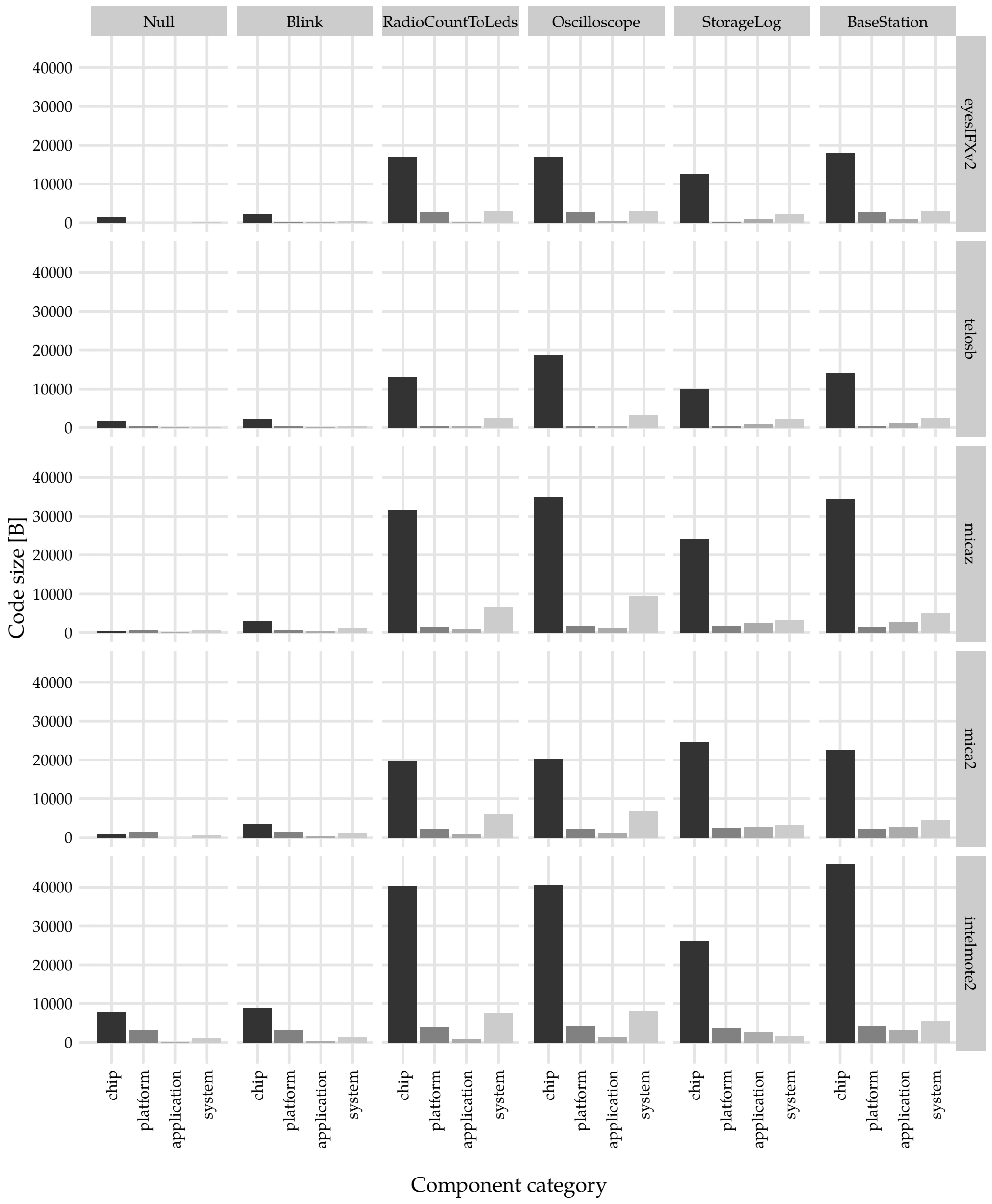


Figure 4.15: Horizontal decomposition of six TinyOS 2.x applications on five hardware platforms: code size.

The results confirm that the majority of algorithmic complexity is concentrated in the chip-specific code. The platform-specific glue is not only smaller in terms of SLOC, it also has very small intrinsic complexity and results in small code footprint. The "non-portability" ratio (platform code size vs. chip and system code size) averages 13.2 % across the different applications and platforms.

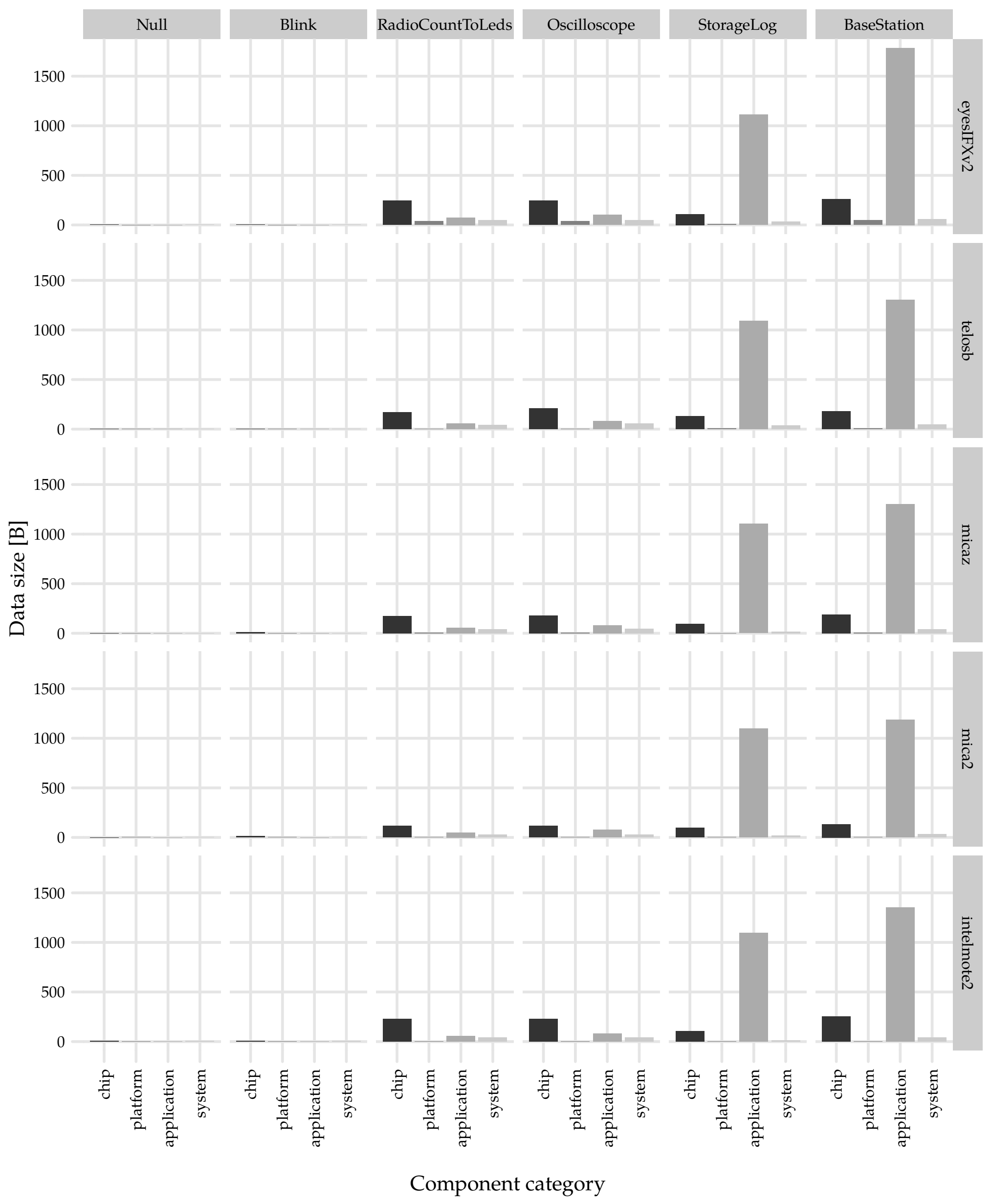


Figure 4.16: Horizontal decomposition of six TinyOS 2.x applications on five hardware platforms: data size.

Memory Footprint

The memory footprint evaluation presented in Figure 4.16, further corroborates the conclusions from the previous two metrics. Apart from the application level buffering that dominates the results in StorageLog and BaseStation, most of the state in the implementations is maintained in the chip-specific components of the portability anchor. In contrast, the platform-specific glue and configuration code is mostly stateless, because it normally just wires the interfaces between the different chips on the platform, without any significant intermediate buffering that would require significant data memory allocation.

Summary

The presented evaluation confirms the successful implementation of the horizontal decoupling principles of the portability anchor in the TinyOS codebase. The selected test applications cover the major hardware subsystems on a typical WSN node. For all of them, and across the three evaluated metrics, the results show high levels of portability, with the platform-independent chip components clearly dominating the platform-specific binding and configuration components.

Given the significant degree of hardware component reuse on the typical WSN platforms (Section 2.2), these facts support our claim that the horizontal decomposition establishes a useful architectural base for a more efficient platform porting process.

4.6.4 Controlling Abstraction Costs

The flexible control over the abstraction costs is core objective for the portability anchor's design and main motivation for separating the hardware-specific abstractions comprising the HAL and the portability wrappers comprising the HIL. In this section, we want to illustrate the need for the decoupling and to show how the two public interfaces at HIL and HAL level facilitate better control over the abstraction costs. In particular we want to demonstrate that:

- HIL provides convenient, platform-independent interfaces for developing portable applications, but the portability may come at a cost in performance and fidelity;
- HAL provides efficient, chip-specific abstractions that offer streamlined access to the underlying hardware resources; and
- using these two public interfaces, the application developer can effectively select the acceptable level of abstraction cost and control the associated portability/fidelity trade-offs.

In the following, we first present our experimental setup, before presenting the used metrics and the evaluation results.

Test Application

To support the above claims, we use a simple test application for servo motor control, a canonical task in many robotic applications which is frequently used to evaluate the fidelity of timer services in real-time OS research [168].

Figure 4.17 illustrates the standard PWM signal waveforms for servo motor control. The period of the control signal is fixed at 20 ms. As the name suggests, the turning

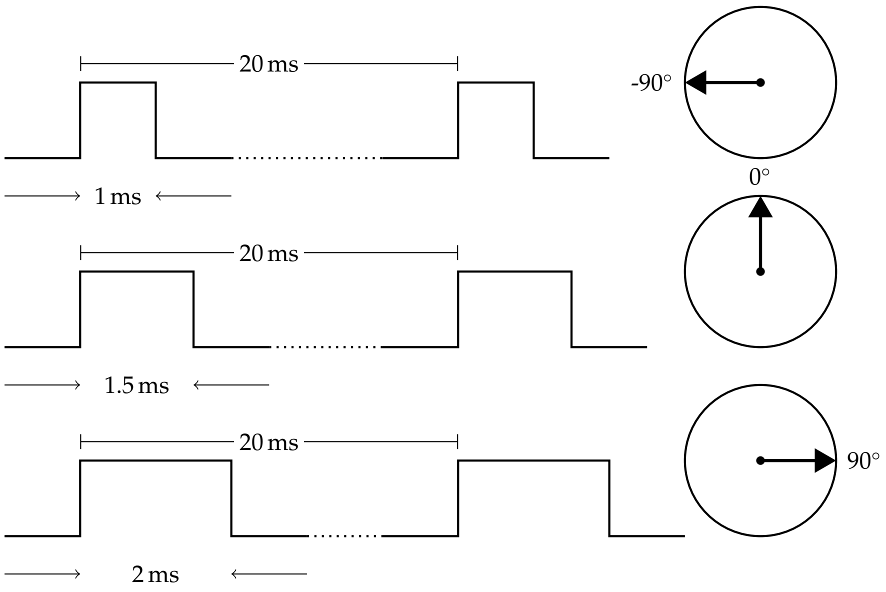


Figure 4.17: PWM signal waveforms for servo motor control.

angle of the motor is controlled via the width of the control pulse at the beginning of the control period. A pulse with a duration of 1.5 ms keeps the servo lined-up (0°). Any shorter/longer pulse than 1.5 ms makes the servo motor turn left/right. The lower and the upper limits are 1 ms and 2 ms, respectively: a pulse with a duration of 1 ms makes the servo turn full left (-90°), and a pulse with a duration of 2 ms turns the servo full right (90°).

The millisecond-resolution of the platform-independent *Timer* interface (Figure 4.9) is insufficient to support the needs of a realistic servo motor control application, where the control pulse duration can dynamical vary between 1 ms and 2 ms. Our goal is to compare the fidelity of the timer abstractions at the HIL and the HAL level, thus, we have constrained the requirements of the test application to the generation of a fixed control waveform that keeps the servo motor turning full right. In other words, the objective is to generate as precisely as possible, a control signal waveform with a pulse width of 2 ms and a period of 20 ms, under moderately high *dynamic load* in the system.

We have based the implementation on the standard *RadioCountToLeds* application (Section 4.6.1), using the telosb platform as a target. To emulate higher dynamic load in the system, we have modified the message sending frequency from the default one message every 250 ms, to sending one message per uniform random interval with maximal duration of 50 ms.

In parallel to the message sending, the application generates the required PWM control signal on a GPIO pin of the MCU. The application is implemented in two variants described in more detail below: a platform-independent variant using only HIL interfaces, and a platform-specific variant that uses a HAL level timer abstraction.

Platform-Independent The platform-independent implementation uses two Timers for timing the period and pulse duration of the PWM signal:

```
module RadioCountToLedsC @safe() {
  uses {
    ...
    interface GeneralIO as ControlPin;
    interface Timer<TMilli> as TimerPeriod;
    interface Timer<TMilli> as TimerPulse;
  }
}
```

On system boot, after initializing the control pin, a periodic *TimerPeriod* is started with a 20 ms period:

```
  event void Boot.booted() {
    ...
    call ControlPin.makeOutput();
    call TimerPeriod.startPeriodic(20);
  }
```

When TimerPeriod fires, the control pin is set high, and a 2 ms one-shot *TimerPulse* is scheduled:

```
  event void TimerPeriod.fired() {
    call ControlPin.set();
    call TimerPulse.startOneShot(2);
  }
```

When TimerPulse fires, the control pin is driven low, which concludes the generation of the pulse:

```
  event void TimerPulse.fired() {
    call ControlPin.clr();
  }
```

The whole process is repeated on the next TimerPeriod firing, thus generating the required control waveform on the GPIO ControlPin.

Platform-Specific In contrast to the portable implementation, the platform-specific variant uses streamlined HAL level access to the timing system abstraction for the performance sensitive parts. Instead of two Timers, this implementation uses two *Alarms* (Figure 4.8):

```
module RadioCountToLedsC @safe() {
  uses {
    ...
    interface GeneralIO as ControlPin;
    interface Alarm<TMilli, uint16_t> as AlarmPeriod;
    interface Alarm<TMilli, uint16_t> as AlarmPulse;
  }
}
```

For easier comparison between the platform-independent and the platform-specific implementation, the *AlarmPeriod* and *AlarmPulse* also have a millisecond-precision like the HIL Timer interfaces. While this is sufficient for generating the fixed 2 ms pulses required by the test application, a real servo motor control code would leverage Alarms with microsecond or 32 kHz (jiffy) precision.

Since the Alarm interface does not support periodic Alarms, on system boot only a single-shot 20 ms AlarmPeriod is started:

```
event void Boot.booted() {
  ...
  call ControlPin.makeOutput();
  call AlarmPeriod.start(20);
}
```

When AlarmPeriod fires, in addition to setting the ControlPin, the Alarm is manually rescheduled, anchored at the last firing time, and a new 2 ms AlarmPulse is started, for timing the duration of the control pulse:

```
async event void AlarmPeriod.fired() {
   call ControlPin.makeOutput()
   call AlarmPeriod.startAt(call AlarmPeriod.getAlarm(), 20);
   call AlarmPulse.start(2);
}
```

Finally, when AlarmPulse fires, the ControlPin is cleared:

```
async event void AlarmPulse.fired() {
  call ControlPin.clr();
}
```

The whole process is repeated on the next AlarmPeriod firing. As a result, the required PWM signal waveform is reflected on the selected GPIO pin of the MCU.

4.6.5 Portability/Fidelity Trade-offs

The above code snippets illustrate how the portability anchor provides convenient domain-specific abstractions for controlling the underlying hardware timers, both at HIL and HAL level. Although their interfaces are very similar, however, the Timers and Alarms encapsulate two very different levels of abstraction, acting as lever for controlling the trade-offs between portability and fidelity in the hardware abstraction code.

As discussed in Section 4.5.3, the portability of the Timer abstraction comes at the cost of higher sensibility to other synchronous code, because its implementation includes deferred processing using tasks which run to completion and can not be preempted by other tasks. The Alarm abstraction, on the other hand, offers a streamlined access to the hardware timers in interrupt context, but at the cost of reduced portability.

To demonstrate the portability/fidelity trade-offs resulting from the use of the HIL and the HAL interfaces, we experimentally evaluated the two implementation variants, focusing on the jitter in the generated control signal, because this ultimately bounds the quality of control over the actuator.

In each experiment, the platform-independent and the platform-specific variants have been evaluated for the duration of 1024 generated control cycles. The experiments have been repeated using five different seeds for the random number generator that controls the message sending frequency and thus the dynamic background load. During the experiment runs, the signal produced on the GPIO pin was captured using a high-speed digitizer and the waveform was post-processed to extract the durations of each period and pulse.

To analyze the jitter in the generated waveform, we have calculated the median and the outer 0.025 and 0.975 quantiles for the extracted pulse and period durations. Figure 4.18 and Figure 4.19 show the results from the analysis of the pulse and the period jitter, respectively. Each point on the graph represents the duration of the pulse/period for a single control cycle. The rectangle depicts the area between the outer quantiles containing 95 % of the readings, while the thick horizontal line is the median duration. The vertical time axis is expressed in binary milliseconds.

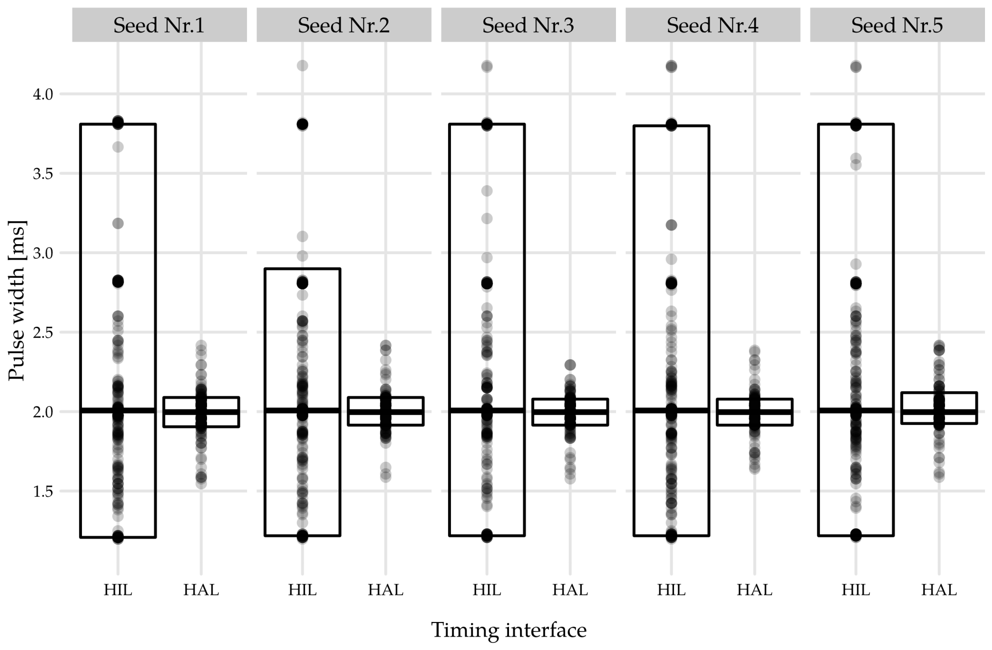


Figure 4.18: Differences in PWM control fidelity: jitter in the pulse duration.

Although majority of the pulse and period durations are clustered closely around the median, in the platform-independent HIL implementation, significant jitter is evident, making precise control of the servo motor almost impossible. We can observe high level of clustering in the durations, indicating that the observed jitter is predominantly caused through the interference between the message sending task and the task in the HIL implementation of the Timers. As the Timer task can not

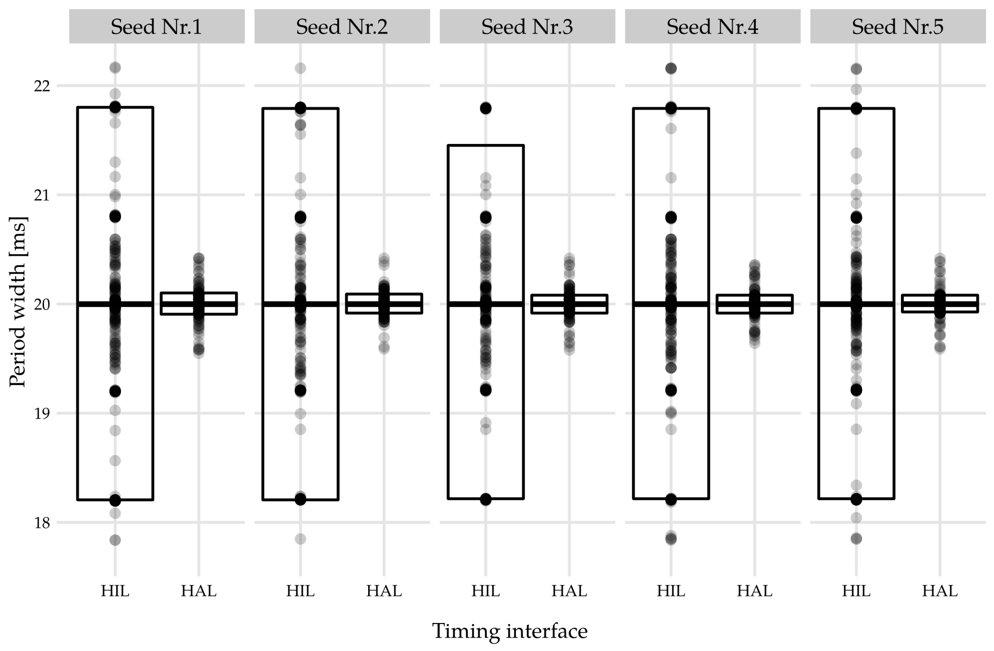


Figure 4.19: Differences in PWM control fidelity: jitter in the period duration.

be run before the other task finishes, this results in delayed TimerPeriod and/or TimerPulse fire events. This in turn, introduces jitter in the duration of the generated pulses and periods.

In contrast, the two Alarms at the HAL level enable much more stable timing, resulting in very good fidelity of the control signal. In this case, the timing is performed in interrupt context. Even if the Alarm expires while message sending tasks are being run, their execution can be interrupted and the time-sensitive alarm rescheduling and GPIO control can be executed without significant delay. The small remaining jitter in the signal is due to atomic blocks in the task implementation during which interrupt servicing is disabled, and the fact that the current msp430 chip implementation prevents interrupt nesting.

The difference in the service fidelity between the two implementations variants demonstrates the need for streamlined access to the underlying hardware in performance-sensitive parts of WSN applications. It shows that changing only a small part of a portable implementation to use the more efficient HAL interfaces can result in significant gains in performance. Switching the PWM signal timing from HIL to HAL level services in the test application required changing only 12 SLOC (most of them substring changes from “Timer” to “Alarm”), representing less than 10 % of the code in the application level components, and a negligible fraction of the total RadioCountToLeds implementation that has more than 10000 SLOC.

CHAPTER 5

Interoperability Anchor

In this chapter we present the design, implementation and evaluation of the *interoperability anchor* of DASA that exports an interoperable CBPS service, while allowing the application designer to adapt the service by making orthogonal choices about the communication components for subscription and notification delivery, the supported data attributes, and a set of service extension components. The framework uses an extended attribute-based naming scheme which is augmented with metadata containing soft requirements for the publishers and run-time control information for the service extension components. It supports different addressing schemes and interaction patterns.

5.1 Design Goals

The internal architecture of the DASA interoperability anchor is driven by the need to customize the generic publish/subscribe model to the specific needs of the WSN domain:

- The standard model fully decouples publishers from subscribers (Section 2.4.5), which means that publishers will often produce data although there are no interested subscribers. Given the tight resource constraints in sensor networks, we believe that publishers should voluntarily be notified of existing subscriptions and given the chance to stop the data gathering process every time a constraint in a subscription cannot be met.

- Due to the large diversity in the needs of the different WSN applications, the publish/subscribe core should be decoupled from the local services like the communication and sensing substrate.

- The application developer should be allowed to customize the aspects of the service with highest performance impact in order to align them with the specific requirements of the target application.
- The implementation architecture needs to be extensible and should allow fine-grained control over the functional and non-functional properties of the service.

In the next two sections we present the main features of the interoperability anchor in a top-down fashion, covering the naming scheme and API as well as the internal decomposition and extension facilities of the component framework. Due to its importance, the discussion on the implications of the decoupling between the publish/subscribe core and the communication protocols is covered in greater detail in Section 5.4.

5.2 Naming Scheme and Service API

Because of the powerful expressiveness, the service interface of the interoperability anchor is based on the Content-Based Publish/Subscribe (CBPS) naming scheme. In contrast to the static *channel* abstraction of the subject-based publish/subscribe, the interests in CBPS are defined as predicates over the whole content of the notification messages [25, 26]. The subscription represents a *content-based address* that *filters-in* the data of interest, and *filters-out* the rest. This results in a very flexible and expressive naming that enables the subscribers to select the interesting notifications with an arbitrary level of detail.

5.2.1 Attribute-based Naming

Theoretically, the CBPS filters can be defined by any function that evaluates to *true* or *false*, when applied to the content of the notification message. In practical systems this freedom has to be somewhat constrained in order to facilitate the effective execution of the matching and the routing/forwarding tasks. One of the most widely used content-based naming subclass is the so called *name/value* or *attribute* system [26]. In this type of naming, the elementary subscriptions are defined as *conjunction* of simple attribute constraints in the form of (*attribute*, *value*, *operator*) tuples. The event notifications, on the other hand, are conjunction of (*attribute*, *value*) tuples.

The various parts of the attribute filter have the following meaning:

Attribute This part of the *content-based address* specifies the property of the notification that will be subjected to inspection during the filtering process. The usage of a preselected set of attributes does limit the expressiveness to a degree, but the complexity of the matching and forwarding tasks is largely simplified.

The attributes are usually typed. Most of the implementations line-up the attribute types to those available in the used programming language. In addition to the basic number and boolean types, almost all of the implementations support string attributes and provide means for extending in the form of User Defined Types (UDTs).

Value The value is selected from the allowed range for the given attribute type and has to be aligned with the operation part of the name, i.e it has to belong to its valid domain. Some of the operations also allow the use of values with *special* meaning like "ANY", "ALL", etc.

Operator The last part of the name tuple contains the binary predicate operator used for the filtering. Primarily, it is one of the common equality and ordering relations ($=$, $>$, $\geqslant$, $<$, $\leqslant$). For the more complex attributes, type-specific operators can be defined (e.g. *substring*, *prefix*, *suffix* operators for strings).

A notification $n = (\mathit{attribute}_n, \mathit{value}_n)$ *matches* a given attribute constraint $a = (\mathit{attribute}_a, \mathit{value}_a, \mathit{operator}_a)$ if and only if:

$$\mathit{attribute}_n = \mathit{attribute}_a \wedge \mathit{operator}_a(\mathit{value}_n, \mathit{value}_a) = \top$$

When a notification n matches a subscription s (i.e. $n \prec s$), it is also said that the subscription *covers* the notification.

5.2.2 DASA Naming and Service API

If a subscription consisted only of constraints over attribute values a subscriber would not be able to explicitly influence the properties of the communication or sensing process like, for example, the sampling rate. Such control properties are conceptually different from the data constraints and can usually not be matched by corresponding *(attribute, value)* tuples in the notification.

We extended the basic naming scheme by allowing subscribers to include *metadata* in subscriptions. Metadata is either exchanged between publisher/subscriber components or plays a key role in controlling service extensions (Section 5.3). It represents control information with soft semantics and is excluded from the matching process.

Metadata is represented by one or more *(attribute,value)* pairs, for example (*SamplingRate, 10*). Conceptually, it represents a notification that the subscriber "attaches" to the subscription. This metadata is specified per subscription and multiple active subscriptions may have different values for the same metadata attribute. Since metadata is non-binding a publisher may apply local optimization techniques: for example, in order to reduce sampling overhead the publisher may decide to combine two subscriptions that address the same attribute by sampling only once with an average sampling rate when the rates are similar, or using the maximal sampling rate when not.

Table 5.1 summarizes the differences between the DASA extended API and the classical publish/subscribe service. The modified naming scheme is supported by two extensions of the basic publish/subscribe service: a "listener" service and a "matching" service. The "listener" service can be used to inform the application

	Basic publish/subscribe API	**DASA publish/subscribe API**

Subscriber	Subscribe($\boxed{C}$)
	Unsubscribe()
	Notify($\boxed{A}$)
		Subscribe($\boxed{C}\boxed{M}$)
		Unsubscribe()
		Notify($\boxed{A}\boxed{M}$)

Publisher	Publish($\boxed{A}$)	Publish($\boxed{A}\boxed{M}$, push)
		Listener($\boxed{C}\boxed{M}$)

Matching		Matching($\boxed{C}$,$\boxed{A}$)

Table 5.1: The basic publish/subscribe service API and the DASA adaptations. A square represents a set of constraints (C), metadata (M) or attribute-value pairs (A). The extended Publish *primitive takes an additional* push *parameter which influences the matching point and is explained in Section 5.4.2.*

about newly arrived subscriptions, which it then can inspect to decide whether to start or stop publishing notifications. The "matching" service may be used by the publisher to check whether a set of attributes disqualifies it from matching a registered subscription. If, for example, the first collected attribute violates a constraint, collecting further data is pointless. When used, these primitives may result in a tighter coupling between publishers and subscribers than in the traditional model, but they have the potential to increase the efficiency of the data collection process, resulting in overall application performance gain.

5.3 Functional Decomposition

These considerations have led to the internal functional decomposition depicted in Figure 5.1 . The publish/subscribe service is distributed and the figure represents an instance of the framework on one sensor node. Each publish/subscribe application is divided into a variable number of *Publisher* and *Subscriber* components. A Publisher component can listen for subscriptions, collect data and publish notifications and Subscriber components can issue subscriptions and receive matching notifications. The *Broker* component provides the publish/subscribe service to the application, it manages the subscription table and it can apply the matching algorithm to filter out notifications that do not match a registered subscription.

The data that the subscribers can subscribe to and publishers can publish is

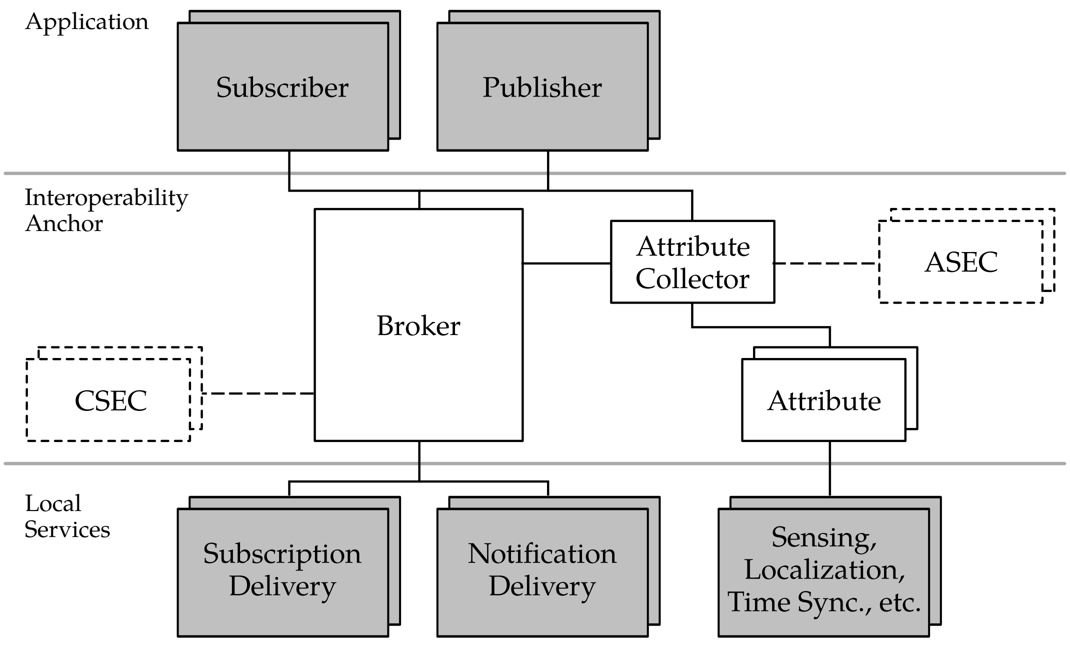


Figure 5.1: High-level functional decomposition of the DASA *interoperability anchor.*

encapsulated in *Attribute* components. In addition to a data collection interface, an Attribute component must provide a matching interface that compares two of its data items based on an attribute-specific operator. The motivation is twofold: first, an Attribute component represents functionality that Publisher components should be able to reuse and access independent of the specific attribute properties. Secondly, matching operators are usually attribute dependent: for example, when sensor readings are affected by hardware-related jitter, the operator "=" should not be interpreted as the exact equality of two values. To increase modularity and keep the core matching algorithm decoupled, this information should be provided by the particular Attribute component.

Within the network, all attributes and operators are represented by integral identifiers. Attribute identifiers are globally unique, while operator identifiers are unique within the scope of a particular attribute. On the edge of the network a translation between identifiers and attribute semantics is performed, for example, using Extensible Markup Language (XML) document maps. The *AttributeCollector* component structures access to the attributes: it maps a request based on the attribute/operator identifier to an actual registered Attribute component.

By including metadata in a subscription a subscriber can influence the communication and sensing process. Often, such control functionality can be isolated in self-contained components for reuse in different applications. For example, a *caching* component could decrease sampling overhead by buffering frequently accessed attribute data when the considered data attribute has high direct sampling costs or is computationally intensive, like feature extraction from acoustic signals. We call such components *Service Extension Component (SEC)*.

A SEC represents reusable functionality that can be plugged into the framework without modification of existing code. A SEC can realize an additional service (as in the caching example) or extend the communication path with additional control information (timestamps, message sequence numbers, etc.). A SEC is associated with one or more dedicated metadata attributes, for example the maximum allowed caching duration, and made available by the application designer *at compile time* (it could even be added at runtime by dynamic over the air code updates). A SEC can be activated dynamically by a subscriber on a per-subscription basis by including an appropriate metadata attribute in the subscription.

The framework supports two different types of extension components, *Communication Service Extension Component (CSEC)* and *Attribute Service Extension Component (ASEC)*. A CSEC can intercept incoming packets before (and outbound packets after) they are processed by the Broker in order to scan the included metadata attributes and, if applicable, perform a specific operation. It can, for example, be used to aggregate notification messages in order to reduce overall network traffic. Since CSECs can also be used to add control information (timestamps, etc.) a subscriber can use the CSECs located on the publisher nodes to (conceptually) assemble its own message header by adding appropriate metadata attributes. ASECs are used analogous for attribute access: they can intercept the requests for attribute data and instead return buffered or processed data dependent on the metadata included by the particular subscriber.

In combination, metadata and SECs realize a soft "control path" in parallel to the basic publish/subscribe "data path". Since SECs are self-contained components and can usually be designed agnostic to data attribute semantics they are easily reusable in different applications and on different platforms. However, when multiple SECs are in use, their ordering must be defined by the application designer because it may influence their overall semantics.

5.4 Communication Decoupling

Any CBPS brokering network has to guarantee the delivery of the relevant notifications to the interested parties. The naive approach would be to *flood* each notification in the broker network, and then locally apply the subscription filters to sort out the relevant notifications from the rest. While stateless, this can lead to wasteful usage of the communication resources as the notifications are also delivered to brokers whose clients are not subscribed.

5.4.1 Integrated CBPS Routing

A more efficient alternative is to perform an integrated *content-based routing* where each broker maintains a *routing table* whose entries associate *filters* and *destinations*.

When a filter F is applied to an notification it evaluates to true or false: $F(n) \to \{\top, \bot\}$. The set of the matching notifications $N(F)$ is defined as $\{n | F(n) = \top\}$. Given this, two filters F_1 and F_2 are *identical* ($F_1 \equiv F_2$) if and only if $N(F_1) = N(F_2)$. The filters are *overlapping* if $N(F_1) \cap N(F_2) \neq \emptyset$, otherwise they are *disjoint*. The filter F_1 *covers* the filter F_2, written $F_1 \supseteq F_2$, if and only if $N(F_1) \supseteq N(F_2)$. The relation is transitive and ($F_1 \supseteq F_2 \wedge F_1 \subseteq F_2$) is equivalent ($F_1 \equiv F_2$).

The integrated CBPS routing leverages these properties of the filters in order to optimize the routing process. In the following we briefly discuss some proposed solutions in this domain, in order of increasing complexity following the classification in [149]:

Subscription flooding The reverse approach to the naive solution is to flood the subscriptions instead of the notifications. This ensures that each broker has *global* information about every subscription in the network. Having this information, the broker network can *guide* the subscriptions only to the interested clients. The result is a reduction in the number of messages that comes at the cost of substantial state in the brokers that severely limits the scalability of the system.

Identity-based Building on the above approach, one simple way to reduce the sizes of the routing tables is to make sure that only one entry is kept for identical filters [148]. This means that a subscription is not forwarded to the neighboring brokers if that was done for an identical one in the past, because they match the same notifications.

Covering-based Further improvements can be achieved by exploiting the covering property to reduce the number of forwarded subscriptions [24, 26]. In this approach, the new subscriptions and unsubscriptions are not distributed to the neighbors if covering ones have already been distributed. Depending on the level of overlap this can noticeably increase the scalability of the system. A significant drawback is the need to resubmit some of the subscriptions when an unsubscription is issued for the covering subscription.

Merging-based Instead of passively examining the relation between the filters, the merge-based model [148] suggests a more active approach where a group of subscriptions are first *merged* in a single covering subscription, and then this single subscription is distributed to the neighbors. In addition to the problem with the unsubscriptions, the merge-routing also requires some heuristic about when and to what extent this merging of the subscriptions is performed.

The performance gains from the above optimizations are dependent both on the level of overlapping between the subscriptions as well as on their spatial distribution in the network. Despite its increased efficiency, the integrated CBPS routing tightly couples the service with the underlying networking infrastructure, making application-specific optimizations very hard.

5.4.2 DASA Routing

The interoperability anchor departs from this tradition and decouples the communication mechanisms from the publish/subscribe core (Fig.5.1). The core broker component has clean interfaces towards the external protocol components, thus *trading some of the optimization potential for increased flexibility* in selecting the subscription and notification protocols.

By exposing the choice of the protocols to the application designer, our framework allows the adaptation of the publish/subscribe service to the specific needs of the application. The type of the communication protocols as well as their energy consumption are likely to have a huge impact on the overall performance, and the application designer should be aware of these implications [85] to make an optimal selection for the particular application. In the following we concentrate on three important aspects of this decoupling and on the architectural features of the framework that address them.

Addressing Support

In contrast to the integrated solutions that rely on a pure content-based routing and forwarding mechanisms, the flexibility of our framework raises the challenge of interfacing with communication protocols that support different dissemination patterns like broadcast, multicast, convergecast, point-to-point, etc., using various addressing models like address-free, id-centric or geographic addressing.

To support this wide range of communication mechanisms we rely on three architectural features. First, the core of the framework is agnostic to the underlying addressing model, and all information relevant for operation of the service is encapsulated in the form of metadata, subscription filters or notification data. Secondly, the interfaces towards the subscription and notification delivery components are kept address-free. Finally, all the addressing information for the communication protocols is provided/consumed by their respective components or wrappers, while the framework provides hooks that facilitate its encapsulation and tunneling when so required. To illustrate this process, we examine the handling of the address information on the subscription and notification path separately.

On the subscription path, a common delivery pattern is one-to-all (broadcast): a subscriber wants to receive notifications from any publisher with matching data in the network. This pattern is naturally supported by the address-free interface. In the case of one-to-many (multicast), the subscriber application defines the scope of the subscription delivery expressed as metadata attribute (hop-count, geographic scope, etc.) inserted in the subscription. The metadata is transparent to the publish/subscribe core and after registration of the subscription in the subscription table its content is passed onto the respective subscription delivery protocol component. The protocol component (or a thin wrapper) extracts the scoping attributes from the subscription content (via suitable accessor functions provided by the core) so that they can be used or translated into corresponding protocol parameters. Depending on the nature

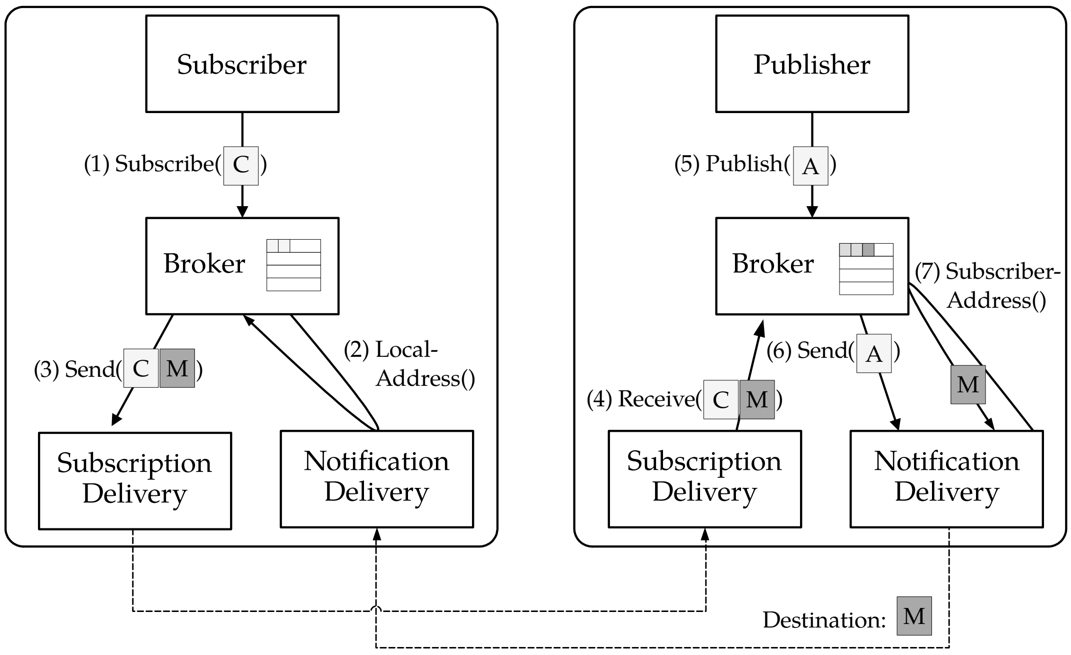


Figure 5.2: Enabling different addressing schemes by tunneling address information as metadata between a subscriber and a publisher. Squares represent constraints (C), metadata (M) or attribute-value pairs (A).

of the scoping parameters, this mechanism might increase the coupling between the subscriber and the subscription delivery protocol, but the publish/subscribe core does not require any adaption. An id-centric, point-to-point subscription delivery, although very atypical communication pattern for a publish/subscribe application, can also be supported with the this mechanism.

On the notification path, the message delivery patterns are potentially more diverse. To abstract from address information and decouple the application from the particular addressing scheme of the notification delivery protocol, we employ the mechanism visualized in Fig.5.2: after a subscription has been issued by the application (1), the notification delivery protocol component (on the subscriber node) can use a hook provided by the core to add the local address of the subscriber as metadata information in the subscription, just before it is disseminated in the network (2).

The addressing information may be expressed using any naming/addressing scheme because the metadata value is transparent to the publish/subscribe core. After the subscription has been disseminated (3) and registered in the subscription tables of potential publishers (4), whenever a notification is published (5, 6) the notification delivery protocol instance on the publisher node can extract the particular source address of the subscriber and use it as address parameter (7).

Thus, the core provides two hooks to the notification delivery protocol: one for attaching the local address to a subscription on the subscriber node and one for reading it out on the publisher node. Both hooks are used optionally – if the notification delivery is, for example, based on flooding or uses data-centric addressing, it will neither add nor read any metadata to/from a subscription. In this way, the Broker, Publisher and Subscriber components remain shielded from the addressing models used by the communication substrate. Even more, the addressing on the subscription and the notification path is decoupled so different addressing models can be used with respect to each other.

At the cost of increased coupling between the core and the communication substrate, the same architecture can even be used to support a "classical" integrated content-based routing protocol. Through single-hop subscription scoping, the core can relinquish complete control over subscription injection and forwarding to the underlying integrated protocol, allowing complex schemes like subscription coverage or merging to be implemented. The resulting duplication of state (subscription table entries, etc.) can be reduced to a certain degree using hooks exported by the core facilitating buffer space sharing. The design of this support is one focus of our future work.

Control of the Matching Point

The departure from the integrated content-based routing and forwarding approach, brings to the surface the question of the "matching point" in the network, i.e. the point where the published notifications are matched against the content filters in the subscriptions. Since the subscription and the notification messages are delivered by potentially separate protocols that do not explicitly share common state, a conscious decision has to be made about where in the network this information would confluence so that it can be passed to the core for matching.

A misplacement of the matching point with respect to the application requirements and the selected communication protocols can result in significant performance penalties as notifications or subscriptions needlessly consume precious networking resources. In general, the optimal location of the matching point depends on many factors like network topology, ratio of publisher to subscriber nodes, frequency of subscription/unsubscription and publication, selectivity and locality of filters, etc.

Our framework supports two major scenarios by default: the filter matching is either applied on the publisher or on the subscriber nodes. Our decision is motivated by several observations. In many sensor network applications, we are faced with either a "pull" or a "push" interaction pattern, i.e. either a small set of subscribers is interested in notifications generated by a much larger set of publishers, or vice-versa, many subscribers are interested in the notifications from a smaller number of publishers. This means that the optimal approach involves either a network wide subscription dissemination with filter matching performed on the publisher nodes or network wide notification dissemination with matching performed at the subscriber nodes [85].

For the cases in between these two extremes, the framework can be extended with a CSEC that determines the optimal points using an integrated content-based routing and forwarding protocol, or from a dedicated "matchmaker" service [67]. The broker component provides a hook that CSECs can then use to execute the matching algorithm, without introducing tight coupling between the underlying protocols and the publish/subscribe core.

Protocol Impact on the Service Semantics

The selection of the subscription and notification delivery protocols is also influenced by the non-functional requirements of the particular application. For example, the application designer may be faced with a scenario where subscriptions need to be updated frequently and not reaching exactly all of the available publishers is acceptable. In this case a protocol for probabilistic best-effort subscription dissemination may be sufficient. On the other hand, an application may require more reliable dissemination of subscriptions and is willing to accept continuous control traffic in the background. In this case a reliable dissemination algorithm would be more suitable. If sufficient resources are available, the application designer might even choose multiple subscription or notification protocols in parallel.

Our framework does not impose any limits on the quality of service provided by the underlying communication protocols, effectively treating them as black box components. Whenever a subscription is issued or a notification is published, the framework will eventually convert the subscription/notification content into payload of the selected protocol component. The choice of protocols therefore has direct impact on the delivery semantics of the publish/subscribe messages, and with that on the semantics of the provided service.

The core itself is not influencing the quality guarantees of the underlying protocols, but SECs can be used to this aim, for example, by periodically retransmitting subscription messages, temporarily storing notification messages, etc.

We contrast the performance and the semantic effects of different types of subscription dissemination protocols in Section 5.6.2.

5.5 TinyCOPS

Assessing the full impact of a component framework like the interoperability anchor of DASA is a difficult task. As with any architecture, the most reliable feedback ultimately comes from surveying programmers after extended periods of day-to-day use.

The development of a reference implementation and its evaluation, however, can be considered as an important first step towards this goal. A real prototype demonstrates that the general design can be implemented under the specific constraints of the target domain. Furthermore, through careful micro-benchmarking executed in controlled, yet realistic setting of modern sensor network testbeds, it provides an opportunity for gaining deeper insight into the specific feature set and the involved design trade-offs.

To this end, we have developed a reference implementation of the interoperability anchor, called *TinyCOPS*, using the TinyOS 2.x (Section 4.5) execution environment.

The internal structure of the TinyCOPS framework closely follows the generic architecture of the DASA interoperability anchor presented in Figure 5.1. In the following we overview the main components and their interfaces in a top-down fashion, focusing on the features involved in the evaluation experiments presented in Section 5.6.

5.5.1 Publishers and Subscribers

The interaction between the applications and the core is abstracted in individual *Publisher* and *Subscriber* components. A Publisher component can listen for subscriptions, collect data and publish notifications and Subscriber components can issue subscriptions and receive matching notifications. The service API provided by these components is shown in Figure 5.3.

```
interface Subscribe
{
  command error_t subscribe(subscription_handle_t handle);
  event void subscribeDone(subscription_handle_t handle, error_t e);
  command error_t unsubscribe(subscription_handle_t handle);
  event void unsubscribeDone(subscription_handle_t handle,error_t e);
  event notification_handle_t notificationReceived(notification_handle_t
      handle);
}

interface Publish
{
  error_t publish(notification_handle_t handle, bool push);
  void publishDone(notification_handle_t handle, error_t e);
}

interface SubscriptionListener
{
  void subscriptionReceived(subscription_handle_t handle);
  void unsubscribed(subscription_handle_t handle)
  command error_t getRegisteredSubscription(subscription_handle_t
      *handle);
}
```

Figure 5.3: The publish/subscribe communication API in TinyCOPS.

The *Publish* and *Subscribe* interfaces are defined in the typical TinyOS split-phase style: an event signals back the result of a command. Both, notifications and subscriptions, are represented by *handles* and realized as abstract data types, and the core provides additional interfaces with operations to create, inspect and manipulate their content (Figure 5.4). The publish/subscribe API is provided in multiple instances, one set for each Publisher/Subscriber component.

A call to *Subscribe.subscribe()* overwrites the previous subscription instance from the same subscriber. A single Subscriber component therefore cannot register more than one subscription in parallel, but it can always modify or unsubscribe this subscription. To issue more than one subscription in parallel an application must instantiate multiple interface instances. On the one hand this means that an application designer must predefine the maximimum number of parallel subscriptions, on the other it increases application robustness, because following the TinyOS 2.x design principle of static allocation, i.e. pushing resource allocation, in this case the subscription table entries, to compile-time.

The *Publish.publish()* command includes a special *push* parameter, which defines whether the notification is disseminated based on the "pull" or "push" interaction pattern. In the first case a matching subscription must have been registered on the publisher node, otherwise the notification is filtered out by the publish/subscribe core. In the second case notifications are disseminated into the network irrespective of any registered subscriptions on the publisher node. Before a notification is signalled to a Subscriber component the matching algorithm is applied (possibly once more) by the publish/subscribe core instance on the subscriber node. This ensures the filtering property of the publish/subscribe API: at any time a subscriber is signalled only the data that matches its current interest.

As motivated in Section 5.2 the *SubscriptionListener* interface (Figure 5.3) can optionally be used to inform the application about newly arrived subscriptions.

5.5.2 Broker and Attribute Collection

BrokerC is responsible for implementing the publish/subscribe service exported to the application. It manages the subscription table allocating one entry for every Publisher/Subscriber component. The BrokerC applies the matching algorithm to filter out those notifications that do not match a registered subscription and it provides functions to access notification/subscription content. BrokerC is also responsible for mapping the CBPS API to different network protocols. In Section 5.5.4 we discuss the specific interfaces and mechanisms in greater detail.

AttributeDispatcherC functions as a switch that decouples from attribute semantics and allows to access attribute data based on attribute identifiers. It provides generic functions to query the size and collect attribute data or determine the matching between attributes and constraints (Figure 5.4).

This enables the BrokerC component to apply the matching algorithm independent of the selected attribute components. In addition, Publisher components can collect sensor data by providing only the attribute identifier and an empty buffer, but without the need to know, for example, the particular attribute metric. Thus, introducing new attributes does not require adaptation of the core code, and publisher logic can be decoupled from attribute semantics.

The AttributeDispatcherC also plays central role in supporting Attribute Service Extension Components (ASECs), introduced in Section 5.3. The component provides

```
interface AttributeCollection
{
  command bool isRegisteredAttribute(attribute_id_t attributeID);
  command uint8_t getValueSize(attribute_id_t attributeID);
  command bool isMatching(avpair_t *avpair, constraint_t* constraint);
  command error_t getAttribute(attribute_id_t attributeID, avpair_t
      *avpair, uint8_t maxValueSize, subscription_handle_t handle);
  event void getAttributeDone(avpair_t *avpair, uint8_t valueSize,
      subscription_handle_t handle, error_t result);
}
```

Figure 5.4: Attribute collection API in TinyCOPS.

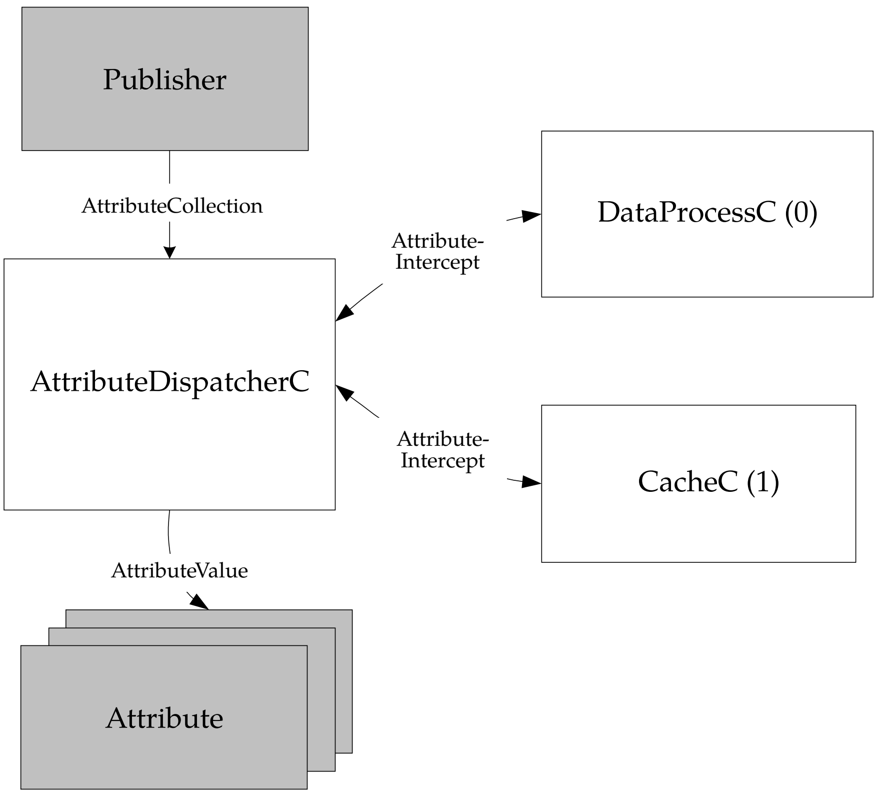


Figure 5.5: Attribute processing using two ASECs, DataProcessC *and* CacheC, *with developer-assigned priorities of 0 and 1.*

means for intercepting the application's request for attribute data and manipulating the returned attribute values. When no ASEC is present, an application's request for attribute data is served directly by the AttributeDispatcherC. When an ASEC is present, it becomes part of the attribute processing loop: it may intercept the request and return buffered or processed data. This action is controlled by the subscriber via respective metadata attributes on a per-subscription basis and by the client of AttributeDispatcherC which can enable/disable ASECs for its call by (not) passing the handle to its subscription. The sequence of interactions for two ASECs is illustrated in Figure 5.5. The ASECs process the request according to the developer-assigned priorities. In the depicted scenario, first *DataProcessC*, and then *CacheC* can intercept and serve the request, otherwise it is served by AttributeDispatcherC with the help of the particular Attribute component. The result is then propagated back, allowing CacheC to update its cache tables and DataProcessC to process the attribute value before it is returned to the publisher.

5.5.3 Attributes

Attribute components define the content that subscribers can subscribe to and publishers can publish. One part of the application design is about selecting the set of Attribute components that will be available for subscription. Every attribute is represented by a unique attribute identifier, which is an integral numbers that can be extracted from subscription and notification content.

The framework supports an arbitrary number of Attribute components. Each Attribute component provides two interfaces: *AttributeValue* to acquire the attribute value and *AttributeMatching* to evaluate an attribute-specific constraint, both depicted in Figure 5.6.

```
interface AttributeValue
{
  command uint8_t valueSize();
  command error_t getValue(void *value, uint8_t maxSize);
  event void getDone(void *value, uint8_t _size, error_t result);
}

interface AttributeMatching
{
  command bool isMatching(const avpair_t *avpair, const constraint_t
      *constraint);
}
```

Figure 5.6: TinyCOPS interfaces for acquiring attribute data and matching attribute-specific constraints, exported by the Attribute components.

Through these interfaces, the Publishers and BrokerC remain fully decoupled from the attribute-specific collection and matching tasks. Leveraging this decoupling, TinyCOPS also includes a generic *StdPublisherC* component that can automatically generate notifications from all registered Attribute components, simplifying the development of application code.

Within a given WSN, all attributes and their operations are identified by an integer number. To map attribute identifiers to their semantic definition, TinyCOPS includes an XML representation for every attribute, which defines the data type, the possible operations or metric conversions as shown in Figure 5.7. Backend applications can use the XML representation to visualize data to the user.

```
<ps_attribute id='0'>
  <attribute_name>EyesIFXTemperature</attribute_name>
  <attribute_type>uint16</attribute_type>
  <attribute_min>0</attribute_min>
  <attribute_max>4095</attribute_max>
  <ps_metric name="Degree␣Celsius">
    <metric_conversion>(X − 1638) / 27.3</metric_conversion>
  </ps_metric>
  <ps_operation id="0">
    <operation_name>=</operation_name>
    <operation_description>equals</operation_description>
  </ps_operation>
  <!—— ... ——>
</ps_attribute>
```

Figure 5.7: An excerpt of the XML representation for the temperature sensor on the eyesIFX platform.

5.5.4 Protocol Components

For each Publisher and Subscriber component, the application developer has to select a network protocol for subscription and notification delivery, respectively. For this, the application developer connects to BrokerC a pair of desired routing protocols.

The framework requires the protocol components to provide a set of standard TinyOS interfaces (Table 5.2). The subscription delivery protocol must provide the address-free *Send* interface (on the subscriber side) and the *Receive* interface (on the publisher side). The interface for the notification delivery protocol must allow a publisher to send a message back to a subscriber via the TinyOS *AMSend* interface. The protocol component must also provide an *Intercept* interface in order to perform, for example, in-network aggregation in CSECs and all protocols must provide the *Get* interface to return the protocol's Active Message ID.

Interface	Required	Provided by	Used by	Role
Send	yes	protocols	core	message delivery
Receive	yes	protocols	core	message reception
Intercept	no	protocols	core	message interception
RootControl	no	core	notification	start beaconing, etc.
Get	no	notification	core	retrieve protocol identifier
AttributeValue	no	notification	notification	acquire local address
PSMessageAccess	no	core	notification	extract destination address

Table 5.2: Interfaces between the TinyCOPS core and the notification/subscription delivery components. The first five interfaces are standard TinyOS interfaces.

The addressing scheme transparency is achieved following the approach described in Section 5.4.2. When the application issues a subscription, by default, BrokerC inserts an additional metadata attribute (*SUSBSCRIBER_AM_ADDR*, N), where N is the node identifier of the subscriber node (its Active Message address).

This happens transparently to the application. On the publisher node, the subscription is registered in the subscription table, and when a notification matches its BrokerC can extract the subscriber address and use it as destination address for the notification message.

When the application wants to scope a subscription to only a subset of the publisher nodes, it can insert in the subscription metadata scoping attributes to define a geographic region. The publish/subscribe core is agnostic to such attributes and will pass the subscription to the specified subscription delivery protocol. Geographic routing protocols must be extended by a thin wrapper component that extracts the geographic address from the subscription and uses them as a protocol parameter. Scoping of subscriptions thus happens transparently to the publish/subscribe core, but requires the Subscriber application and wrapper component to agree on the same metadata attributes.

Table 5.3 summarizes what protocols can be used for subscription and notification delivery in a TinyCOPS application, based on their addressing scheme. TinyCOPS does not impose requirements on the quality of service message delivery, rather it is the task of the application designer to choose protocols that match the application requirements.

Addressing scheme	**Subscription delivery protocol**	**Notification delivery protocol**
address-free	supported	supported
id-centric	not supported	supported
data-centric	supported	supported
geographic	supported	not supported

Table 5.3: Supported protocols for subscription and notification delivery in TinyCOPS, based on their addressing scheme.

If multiple protocols are used in one application TinyCOPS can ensure that publishers and subscribers can interact only if they use the same set of protocols. For this purpose the core component can insert in every subscription an additional metadata attribute that identifies the notification delivery protocol by its protocol identifier. On every potential publisher node the core will extracts such a metadata attribute and signal the subscription to a local Publisher component only if it has specified the same set of protocols. If this is desired, the notification protocol must provide the *Get* interface returning its protocol identifier.

5.5.5 Application Composition

Creating a TinyCOPS application involves making decisions about the data attributes and possible SECs, the number of Publisher/Subscriber components and the respective protocols. Figure 5.8 shows the code for an example publisher application that uses two Attribute components. The application leverages the generic StdPublisherC component introduced in Section 5.5.3, that listens for a subscription and automatically publishes corresponding notifications by querying the AttributeDispatcherC for any necessary attribute data. The application uses the publish/subscribe service via a wrapper,*TinyCOPSClientC*, that structures the access to the core. *DisseminationTrickleC* and *CtpWrapperC* are wrapper components for protocols used for subscription and notification delivery.

```
configuration TinyCOPSDemoAppC
{
}
implementation
{
  components new StdPublisherC() as Publisher,
             // the publisher application
             new TinyCOPSClientC() as Client,
             // structured access to the broker
             new CtpWrapperC() as NotificationProtocol,
             // notification protocol
             new DisseminationTrickleC() as SubscriptionProtocol;
             // subscription protocol

  components AttributeTemperatureC, AttributePingC;

  // wiring (specifying) the protocols for the publisher
  Client.ReceiveSubscription -> SubscriptionProtocol;
  Client.GetSubscriptionAMID -> SubscriptionProtocol;
  Client.SendNotification -> NotificationProtocol;
  Client.ReceiveNotification -> NotificationProtocol;
  Client.InterceptNotification -> NotificationProtocol;
  Client.GetNotificationAMID -> NotificationProtocol.GetAMID;
  Client.PacketNotification -> NotificationProtocol.Packet;
  Client.PacketSubscription -> SubscriptionProtocol.Packet;
  Client.RootControl -> NotificationProtocol;

  // wiring the service interfaces to the publisher
  Publisher.Publish -> Client;
  Publisher.SubscriptionListener -> Client;
  Publisher.PSMessageAccess -> Client;
  Publisher.PSHandle -> Client;
  Publisher.AttributeCollection -> Client;

  // the attribute components are auto-wired
}
```

Figure 5.8: Creating a simple publisher application with TinyCOPS.

5.6 Evaluation

In this section we leverage TinyCOPS, as a mature prototype of the DASA interoperability anchor architecture, to demonstrate the degree of achievement of the major design goals of providing a high-level service for rapid development of WSN applications, while maintaining flexibility and offering possibility for application-specific optimization of the service.

TinyCOPS also corroborates the evaluation of the portability anchor presented in Section 4.6, since it demonstrates that the lower anchor in the DASA architecture provides solid base for developing complex portable services. In Section 5.6.2, we illustrate the seamless operation of the framework on different hardware platforms, facilitated by the platform-independent services of the portability anchor's implementation in the TinyOS 2.x code-base.

5.6.1 Resource Usage

In this section we briefly evaluate the local static and dynamic resource usage in different TinyCOPS components, following a similar approach as the one presented in Section 4.6.1. We report on the code size required by the components in a typical TinyCOPS application as well as the processing time overhead of the core component. The results confirm that our framework introduces acceptable code, memory and execution time overheads.

The evaluation of the code size footprints presented in Table 5.4 confirms the broker component as the most complex element. By offloading the complexity in the broker component, the framework allows composing lean applications. This is demonstrated by the size of the *StdPublisherP* component, a generic Publisher component included in TinyCOPS for convenience. The component simply listens for a subscription and publishes corresponding notifications by querying the *AttributeCollector* for attribute data. It is agnostic to the attribute semantics and can respond to any subscription as long as the respective *Attribute* components has wired to the *AttributeCollector*. The *Send-On-Delta* CSEC (introduced in Section 5.6.2) can handle attributes of different integer sizes, a flexibility that is paid in increased program memory footprint.

To get an insight in the processing overhead introduced by TinyCOPS we measured the code execution time for the subscribe and publish operations on a telosb mote. We used an application that subscribes to a single attribute and measured the time it takes for a subscription/notification message to pass through the TinyCOPS core and protocol wrapper components (using Collection Tree Protocol (CTP) as dissemination protocol). Under this scenario, the main tasks of the core were management of the subscription table and performing the matching algorithm. With a CPU operating frequency of 4 MHz, the subscription send-path requires 144 µs and the subscription receive-path 281 µs. For notifications, it took 127 µs on the send-path and 88 µs on the receive-path. For comparison, the time between posting a task and executing it takes 48 µs. While the results are dependent on the time spent for matching an attribute-value pair with a constraint (we used the TinyCOPS *Ping* attribute), there are no additional "deferred" costs involved (for example, posting tasks or setting timers for later execution).

Creating a TinyCOPS application involves making decisions about the Attribute

Component Name	Description	SLOC size [B]	Code size [B]	Data
BrokerImplP	Broker	671	2838	19
SendOnDeltaP	Send-On-Delta CSEC	103	1126	12
StdPublisherP	Publisher	180	738	70
SubscriberGWImplP	Subscriber (+ gateway)	138	554	129
AttributeCollectorP	AttributeCollector	87	154	4
CSECDispatcherImplP	CSEC "Glue"	115	98	2
Msp430InternalTemperatureP	Temperature Attribute	20	2	-

Table 5.4: Code and memory footprints of an example TinyCOPS application with one Publisher, one Subscriber, one CSEC and one Attribute component.

as well as service extension components, the number of Publisher/Subscriber components and their respective communication protocols. Because in TinyOS an application is created by *wiring* components together, and since Attribute and service extension components can be designed to self-wire to the TinyCOPS core, a single line of code suffices to make them part of an application, respectively. The wiring part of a typical TinyCOPS application as described in Section 5.6.2 consists of about 32 SLOC, excluding the actual Publisher/Subscriber logic and the protocol wrappers. As a reference, the dissemination protocol wrapper consists of 97 SLOC.

5.6.2 Distributed Testing with TWIST

To demonstrate the full benefits of the specific architectural features of the DASA interoperability anchor, the performance of TinyCOPS has to be evaluated in the natural distributed context in which normal WSN applications operate. To this end, we have designed a flexible distributed testing infrastructure, the TKN Wireless Indoor Sensor Network Testbed (TWIST), which we describe in detail in Chapter 6. The results presented in this section were obtained using the instance of TWIST at the Telecommunication Networks Group (TKN) office building on TU Berlin's campus (Section 6.4.1).

Starting with a simple data collection application scenario, we present experimental results which show that the choice of dissemination protocols can exhibit considerable performance tradeoffs (Section 5.6.2). We then gradually increase the complexity of the application. Section 5.6.2 describes the integration of a send-on-delta service extension component and the effects on application performance and in Section 5.6.2 we show how TinyCOPS is used to extend the application with an alarm notification service realizing both "pull" and "push" interaction pattern at the same time.

We have opted against head-to-head comparison of TinyCOPS with other monolithic publish/subscribe frameworks because the overall performance of the frameworks is dominated by the underlying protocols and not the architectural features, there is currently no TinyOS 2.x implementation of a monolithic publish/subscribe framework that would facilitate direct comparison, and even if such an implementation was available, the comparison results would be vulnerable to differences in the invested optimization effort. Instead, the evaluation scenarios are focused on demonstrating the flexibility and versatility of the design. The obtained results corroborate our claims that the framework:

- exports significant performance trade-offs to the application in an easy-to-use fashion; and
- is general and flexible enough to support different interaction patterns;

Tradeoffs in Protocol Selection

To demonstrate the tradeoffs that TinyCOPS exposes to the application designer through protocol selection we contrast two subscription delivery protocols: a plain flooding protocol (every node that hears a subscription broadcasts it to all its neighbours once) and an epidemic broadcast protocol. The latter is part of the TinyOS 2.x core and based on the Trickle algorithm [126, 128]: it lets nodes *continuously* broadcast status information about the subscriptions they have received. Whenever a node hears an older subscription than its own, it broadcasts an update to its neighbors. In contrast to the flooding protocol, which ends its operation after a short time, the epidemic protocol (called "TinyOS 2.x Dissemination") remains active.

We created a simple TinyCOPS application with one subscriber and the rest of the nodes used as publishers. In our first measurement we disseminated the subscription via plain flooding. In the second, we used the TinyOS 2.x Dissemination protocol. The modification is done by changing a single line of the TinyCOPS application configuration. For notification delivery, in both measurements, we use the TinyOS 2.x Collection Tree Protocol (CTP) [71] performing best-effort, multihop delivery of notifications to the sink of the tree (subscriber).

Both measurements lasted 90 minutes and were made with 86 Tmote Sky nodes, 85 publisher nodes and one subscriber (used as basestation, bridging to/from a host computer). At time t_0 a subscription was injected asking for notifications to be published with a rate of one notification per minute by each publisher. After 30 minutes, at time t_1, one third of the publisher nodes (randomly chosen) were shut down and 30 minutes later, at time t_2, they were powered up again. Nodes that were shut down lost all state including subscription table entries.

Figure 5.9 shows the percentage of active publishers over time. We define active publisher as a node that has registered a subscription and published at least one notification. At time t_1 the number of active publishers decreases by about 30% due to our active power management. The difference between the protocols becomes visible at time t_2 when these nodes are powered up again: the epidemic Dissemination protocol quickly manages to spread the subscription to the recovered nodes, while the flooding protocol cannot (the subscription was injected only once at time t_0).

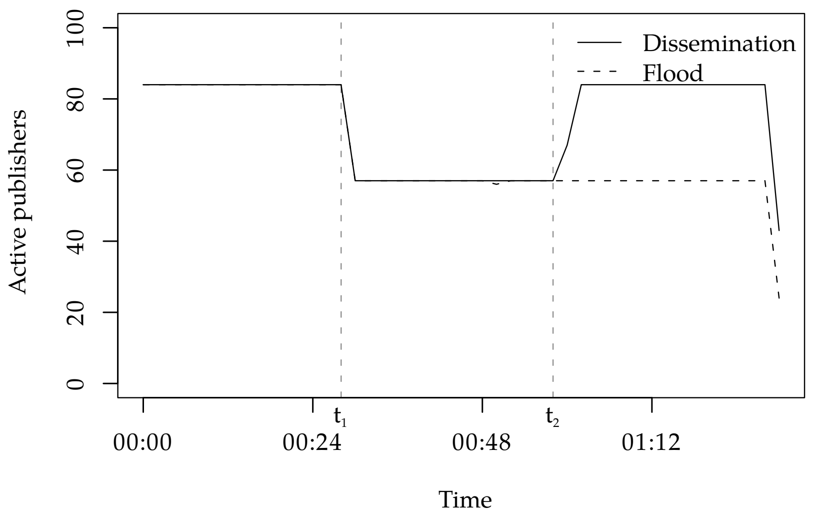


Figure 5.9: Number of active publisher nodes.

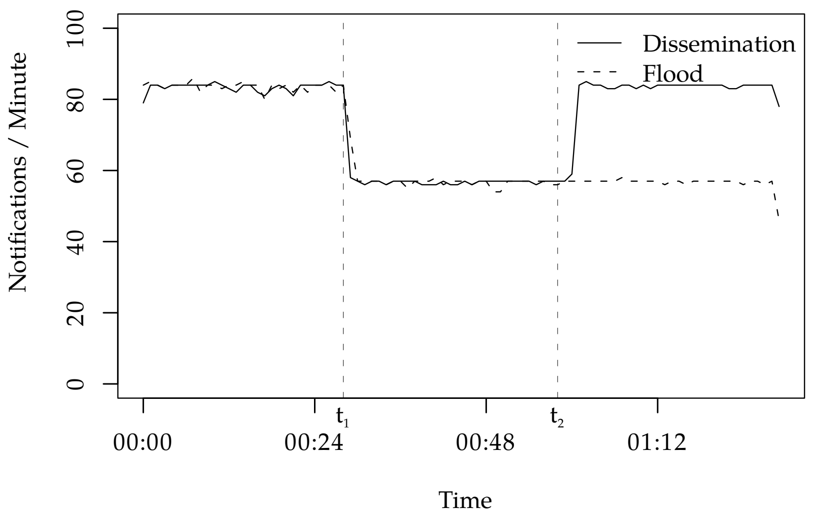


Figure 5.10: Notification goodput using TinyOS 2.x Dissemination and flooding for subscription delivery.

Figure 5.10 shows the changes in notification goodput perceived by the subscriber. We define notification goodput as the number of distinct notifications that arrive at the subscriber in a fixed time window of one minute. The curves almost match the number of active publishers and indicate a very good delivery ratio of CTP.

We used the serial backchannel of the testbed to let all nodes periodically output status information about the number of different messages they had sent over the wireless channel. This information allowed us to derive the traffic for subscription delivery as depicted in Fig.5.11. The figure visualizes the tradeoff between the protocols: the flooding protocol generates one message for each node in the network at the time the subscription is injected. The Dissemination protocol generates more messages, but is able to update the rebooted publishers at time t_2.

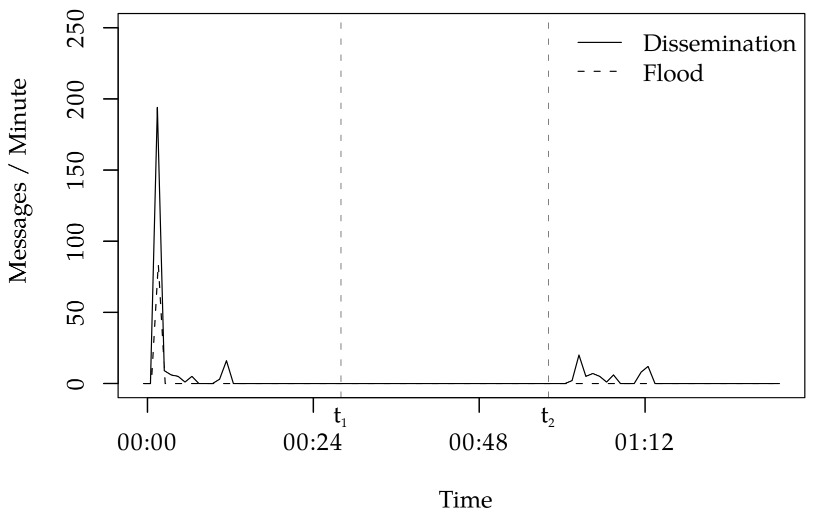


Figure 5.11: Subscription delivery protocol traffic.

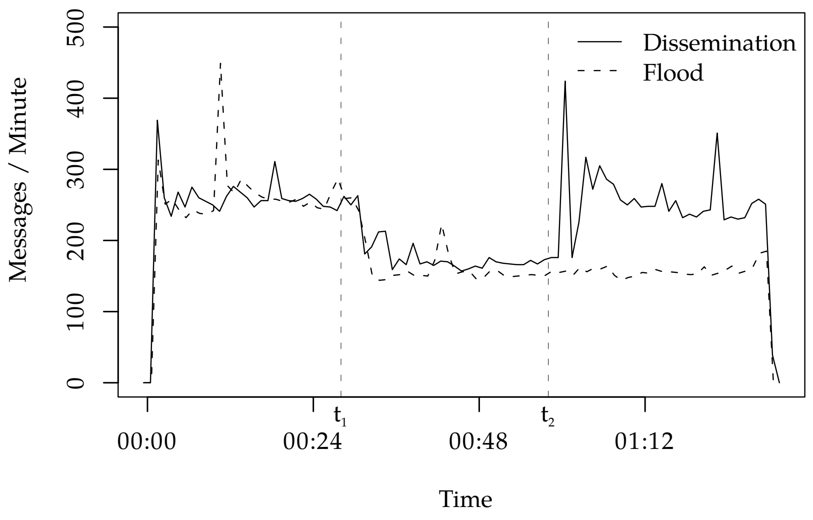


Figure 5.12: Notification delivery protocol traffic.

Finally, our setup allowed us to determine the number of notification messages sent in the network by all nodes over a time window of one minute (Fig.5.12). On average 3 messages were sent per notification, however our setup did not allow us to differentiate between retransmission and forwarded messages.

Adding a Service Extension Component

To decrease notification traffic and effective energy consumption, we modified the baseline application described in the previous section to realize a "send-on-delta" approach: notifications should be published only if the attribute values deviate by more than Δ from the previously published notification. Δ is defined by the subscriber and specified as the metadata of the subscription. To make the functionality resuable we implemented it as a CSEC *SendOnDeltaC* that intercepts outgoing notifications. It maintains a buffer for the last published notification, calculates the difference between the attribute values and suppresses the publishing if the difference is smaller than specified in the corresponding subscription. It is agnostic to attribute semantics and can be used for any attribute with integral data type.

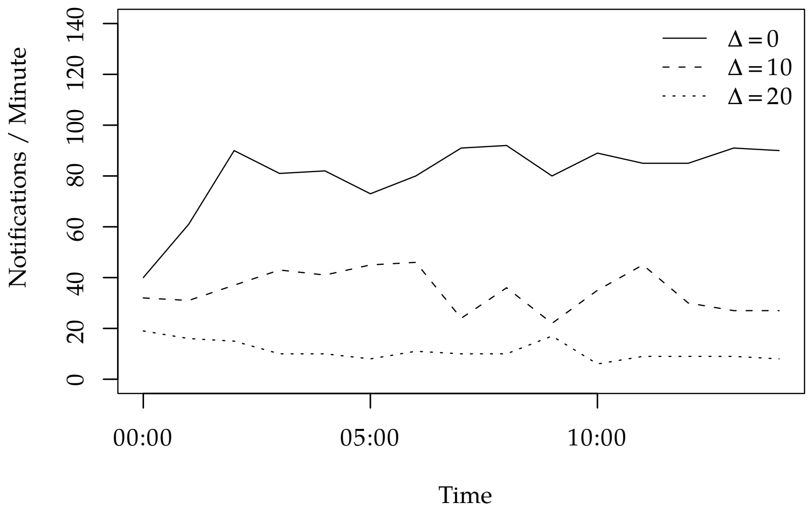


Figure 5.13: Effects of a varying Δ on notification goodput.

We performed three measurements with 86 eyesIFX nodes and varying Δ and observed the effects on notification goodput as perceived by the single subscriber. The subscription asked for light sensor data to be published with a rate of one minute by each publisher and Δ was chosen 0, 10 and 20, where 0 means that all notifications are published and 10 and 20 represent the Δ of luminosity in absolute values of the raw eyesIFX light sensor reading. Figure 5.13 shows the effect on notification goodput: with a higher Δ, more notifications are suppressed by the CSEC, giving to the subscriber application a powerful runtime control over the tradeoff between data resolution and communication overhead.

Creating a Combined Push and Pull Application

Previous work [85] has shown that the interaction pattern between publishers and subscribers ("pull" vs. "push") can significantly affect application performance and should be carefully aligned with the ratio of publishers to subscribers. We created an application that included two Publisher components, one for periodic temperature data collection and one for generating fire alarm messages. We wanted the fire alarm event to quickly propagate to all rooms of the office building, but periodic measurements to be collected only by a single subscriber. We therefore selected a single node to disseminate a subscription which notifications from the first Publisher component had to match (locally, based on the "pull" model). Fire alarms, however, were "pushed": whenever the second Publisher component detected a fire alarm regardless of any registered subscription, it immediately distributed the notification to all nodes in the network. The first Publisher component was "wiring" the subscription delivery protocol to the core and using CTP for notification delivery. The second Publisher component "wired" the flooding protocol for notification delivery.

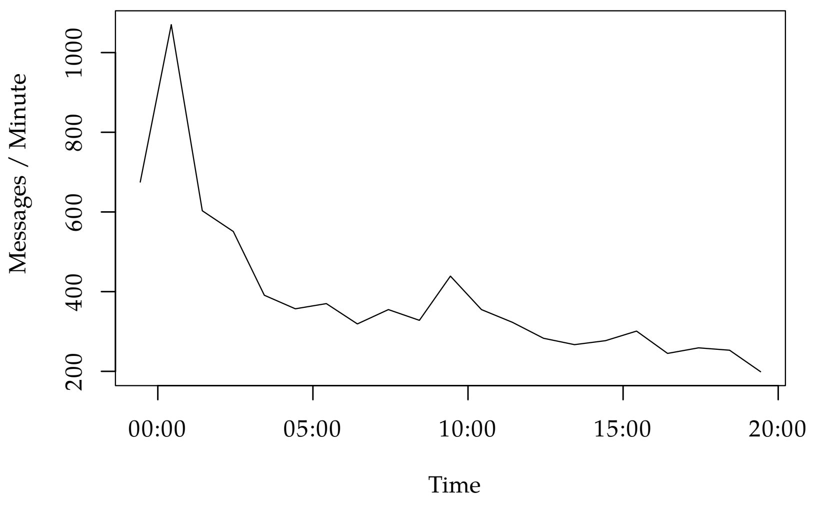


Figure 5.14: Example of a "pull and push" interaction.

Figure 5.14 shows a trace of the communication rates collected over 20 minutes on 85 Tmote Sky nodes. It represents the total number of packets sent by all nodes for a fixed time window of one minute. One subscription for periodic data collection is issued at the start of the measurement using the TinyOS Dissemination protocol, 10 minutes later we simulate a fire alarm, by sending a serial packet to one of the publisher nodes (randomly chosen). This node then started a flood of notification messages. The increase in traffic is visible by a small spike, however, it is almost masked by the high level of CTP "pull" traffic.

5.7 Related Work

In [119], the SPIN family of protocols is presented, that use metadata-based negotiation phase to protect the network resources from unnecessary data exchanges. The content filtering capability in our framework has the same goal. In our case, the metadata part of the subscription message is used to convey a set of "non-binding" requirements from subscribers to publishers while the constraints express the imperative filtering. In SPIN, the metadata format is considered to be application dependent. We believe that the attribute-based naming scheme is flexible enough to support the majority of data-driven applications. Having a fixed naming scheme helps in optimization of the matching components and improves the portability of the application code.

Our naming scheme is much closer to the one used in the Directed Diffusion family of protocols [100], but we make clear distinction between metadata and constraints and support attribute-specific operators. Conceptually, however, more important is the difference in the level of decoupling between the middleware service implementation and the communication protocols. Our framework not only delineates cleanly at this interface, it also allows for individual customization of the subscription and the notification delivery protocols and provides infrastructure for address information tunneling and matching point control.

MiLAN [87] is a flexible sensor networks middleware that continually tracks the application needs and performs run-time optimizations of the network and sensor stacks to balance the application QoS and the energy efficiency. It is positioned as a general framework that can also be used with resource rich wireless technologies like IEEE 802.11 and Bluetooth. Our framework is concentrated on the class of relatively resource limited sensor network hardware where compile-time optimization has comparably large impact, and where the run-time modifications are mostly limited to parameter tuning.

The Mires middleware [188] provides a publish/subscribe service, but uses the component architecture of TinyOS 1.x. Mires uses a topic-based naming scheme that lacks the expressiveness of the content-based filtering. While Mires envisions the possibility of introducing new services (like aggregation) using extension components, the choice of the communication protocols is fixed and can not easily optimized to the needs of the application.

Our aim to increase the flexibility of the framework also has parallels with the existing work on versatile publish/subscribe systems [55]. Similarly, the concept of adding metadata to the constraints, and filters in the notifications is currently being investigated by the "mainstream" publish/subscribe research community [199].

5.8 Summary

A major design goal of the DASA interoperability anchor is to decouple the service sub-tasks which are expected to have large impact on the resource usage. This decomposition strives to give an application designer a simple and flexible means to select protocol components and data attributes according to his needs, and to give him more fine-grained control over the publish/subscribe service through the concept of extension components.

With TinyCOPS, we provide a reference implementation of the DASA interoperability anchor that is aligned with the design philosophy of TinyOS 2.x. Using this prototype, we have experimentally demonstrated the flexibility of the DASA design and its ability to support different sensor node platforms, communication protocols and interaction patterns. On the example of a "send-on-delta" service extension component, we have illustrated how the framework can be augmented in order to give the application designers additional control knobs for trading-off different performance objectives.

CHAPTER 6

Distributed Testing Infrastructure

In this chapter we introduce the design of TWIST, our testing framework that enables efficient testing of functional and non-functional properties of distributed WSN services under realistic but controlled settings. TWIST supports multiple SUT platforms, out-of-band signaling, powerful topology control and fault injection capabilities. All these features were instrumental for the evaluation of the DASA interoperability anchor prototype presented in Chapter 5.

In addition to TWIST, we also introduce the design of a novel testbed federation platform that supports standardized experiment specification and easy repetition of experiments across different WSN testbeds. The platform simplifies the realization of cross-validation studies which are essential in differentiating between the intrinsic properties of the SUT and the effects of the testing environment.

We conclude the chapter by presenting our local instance of TWIST, deployed in the TKN office building at TU Berlin's campus. We report on the deployed hardware and software infrastructure, and we evaluate the achievement of our design goals through analysis of operational data.

6.1 Design Validation and Testing

The configuration freedom offered by component-based architectures like DASA, can only be exercised safely if it is accompanied by means for verifying that the intended semantics of the service interfaces has not been violated during the configuration process. For example, the flexibility in optimizing the communication substrate in the interoperability anchor in DASA, has to accompanied by means for checking the resulting semantics of the exported CBPS service.

One way to rigorously validate the composed system is through *model checking* [33]. The approach relies on exhaustive exploration of the execution space of an approximating finite state machine that models the behavior of the system and its comparison with a set of invariants typically specified using temporal logic. Unfortunately, in systems with many richly interacting components, the model checking is faced with a state explosion problem. In many realistic scenarios, one has to resort to simplifying assumptions about the behavior of the components and their interaction in order to keep the computation tractable [64], which diminishes the relevance of the obtained results. The large scale, the heterogeneity, and the rich coupling with the environment, make realistic model checking of real WSN systems prohibitively complex.

While not guaranteeing exhaustiveness, the *testing* approach represents a valuable design validation alternative. Testing is particularly suitable for validating the composed system in the case of "black-box" reuse, when the internal implementation details in the components remain opaque to the system composer [63]. Due to their distributed nature, the testing of the WSN systems requires distributed testing infrastructure, in the form of *testbeds*, which allows realistic and controlled experimentation with the SUT.

The specifics of the WSN domain pose significant challenges for their design and implementation. In particular, effective WSN testbeds have to:

- approach scales similar to typical WSN deployments and replicate the target environment as much as possible;
- support large number of heterogeneous, resource constrained SUT platforms;
- provide sufficient isolation between the testing infrastructure and the SUT, because the SUT operation on the resource constrained WSN platform can be easily affected and disrupted by the testing process;
- support emulation of topology changes and injection of failures, allowing reliability and robustness testing of the SUT; and
- support coordinated injection of commands and extraction of experimental data over the testing infrastructure, allowing easy distributed debugging and causality analysis with minimal impact on the SUT operation.

These barriers have resulted in scarcity of adequate testing infrastructure, hindering the research and development of WSN systems. Motivated by their importance, we have designed and developed TWIST, a flexible distributed testing infrastructure that enables efficient testing of functional and non-functional properties of distributed WSN services.

6.2 TWIST Testbed Platform

Figure 6.1 depicts the high-level architecture of TWIST. In the following we describe the individual testbed entities, starting from the lowest layer—the sensor nodes—and moving up to the testbed backbone with the attached server and control station.

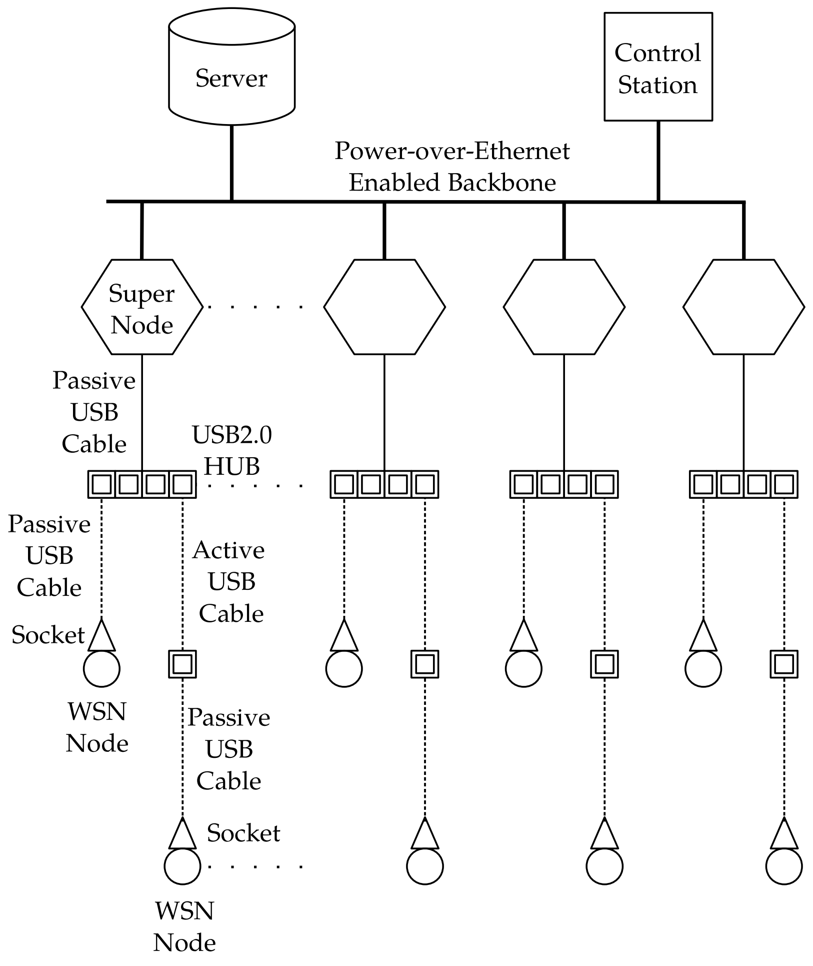


Figure 6.1: Hardware architecture of the TWIST testing platform.

6.2.1 Sensor Nodes

As main hardware units of the SUT the sensor nodes need a set of hardware capabilities facilitating their seamless integration with the rest of the testbed infrastructure.

They have to expose suitable hardware interfaces that support external powering, reprogramming, as well as out-of-band exchange of configuration, debug and application data. Traditionally, these functions have been served by dedicated interfaces: power supply bus, JTAG for programming and debugging, RS-232 for serial communication, etc. This was significantly complicating the hardware and software integration between the SUT and the tester, making customized hardware solutions necessary, and driving the costs high.

Recently, several WSN node platforms using USB as standardized hardware interface have emerged (Section 2.1.1). Thanks to the features offered by the open USB specification and the flexibility of the current UART-to-USB chips, this hardware interface is able to replace almost all of the custom interfaces, simplifying integration of the sensor nodes with external systems. This migration to a single standardized interface drastically lowers the costs for the testbed due to the reduced prices of the components as a result of the economies of scale.

The overall architecture of the TWIST is crucially centered around the use of the USB interface. We are able to support a heterogeneous mixture of WSN platforms as long as they export the above listed capabilities (power-supply, programming and communication) via a standard-compliant USB interface. For example, both the eyesIFX and the telos mote families (Section 2.1.2) satisfy the above conditions and have been successfully interfaced with TWIST.

In addition to having a compatible interface, the sensor nodes need enough slack computational and memory resources to handle the load imposed by the interactions with the testbed. Only in this way the internal state of the SUT can be monitored and influenced without disturbing the normal execution of the application.

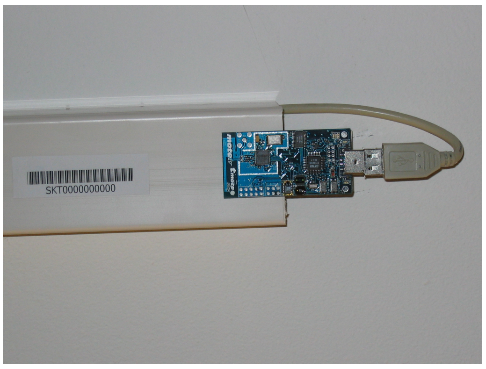
On the software side, the operating system running on the sensor nodes has to satisfy several basic requirements. First, it has to provide a suitable execution environment for the application logic of the SUT. Secondly, it should support node configuration, instrumentation of the application code and allow for out-of-band communication with the super nodes over the USB infrastructure.

Both TinyOS versions satisfy these requirements and run on both the telos and the eyesIFX platforms. It provides a generic and lightweight execution platform for sensor network applications. TinyOS is already shipped with components that support communications over the serial interface (serial-to-USB converter) using a protocol similar to Point-to-Point Protocol (PPP)/High-Level Data Link Control (HDLC). On top of this protocol, another TinyOS component enables `printf()`-like logging as well as bridging debug and application messages. Using tools like Marionette [215], even more powerful, Remote Procedure Call (RPC)-like interactions can be supported.

It is important to note that from the perspective of the super nodes it is not necessary to have TinyOS running on the sensor nodes. Any execution environment implementing a mutually agreed communication protocol between super nodes and sensor nodes will suffice.

6.2.2 Testbed Sockets and USB Cabling

Seen as plain hardware, a testbed *socket* is nothing more than the point where the USB interface of the sensor node attaches to the USB infrastructure of the testbed. The architectural significance of this point is, however, much greater. The sockets have unique identifiers, and their geographical position is known and does not change over time. We associate the node identifiers to the socket identifiers and hence to the geographic position of the sockets. The sockets are connected to the remaining testbed using a combination of passive and active USB cables, depending on the distance between the socket and the next element of the infrastructure – the USB hubs. Using passive cables a maximum distance of 5 m can be bridged. For greater distances, "active USB cables" can be used (single port USB hubs with fixed cable), or several USB hubs can be daisy-chained together.


(a) telosb mote in a testbed socket

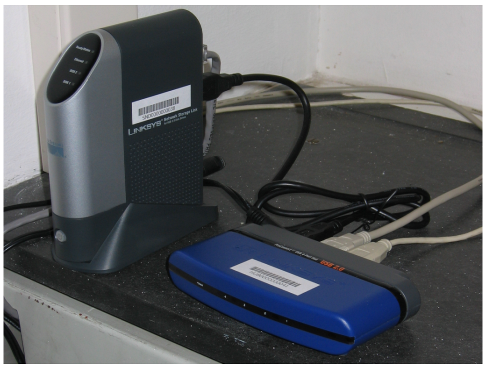

(b) Super node and USB-hub arrangement

Figure 6.2: Instances of TWIST hardware.

6.2.3 USB Hubs

The hubs are the central element of the TWIST USB infrastructure and provide support for some of the most important features of TWIST.

At the most basic level, the USB hub is a multiplexing device that enables us to break the one-to-one correspondence between the sensor nodes and the second-level testbed devices which can be found in many of the existing WSN testbeds. This enables significant cost savings without compromising the testbed functionality. Even more, the USB hubs give TWIST one of its most powerful capability: the binary power-control over the sensor nodes in the testbed.

The USB Hub Specification 2.0 requires that self-powered hubs support port power switching. By sending a suitable USB control message, the software can control the power state of a given port on the hub, effectively enabling/disabling the power supply for any attached downstream device. In the case of TWIST, these downstream devices are the sensor nodes plugged into the testbed sockets. This means that we are able to individually control the power supply of any sensor node in the testbed by simply issuing a suitable USB control message.

Depending on whether the sensor node attached to the socket has a battery or not, this enables four different power-supply transitions:

- from "USB-powered" state into "off" state;

- from “off” state into “USB-powered” state;
- from “USB-powered”state into “battery-powered” state; and
- from “battery-powered” state into “USB-powered” state

6.2.4 Super Nodes

If TWIST only relied on the USB infrastructure, it would have been limited to 127 USB devices (both hubs and sensor nodes) with a maximum distance of 30 m between the control station and the sensor nodes (achieved by daisy-chaining of up to 5 USB hubs). While suitable for small to medium size testbeds [201], these limitations are not compatible with our goals of scalability of the architecture and support for deployments over large geographical areas.

In itself, the USB length problem can be tackled with various range extension solutions: using custom signal-enhancing devices, converting to another long-range serial protocol like EIA-422 or EIA-485, or completely changing the communication technology by moving to USB over Ethernet, etc. Nevertheless, all these solutions still concentrate the complete responsibility for the testbed functions in a single host, severely limiting the scalability of the architecture.

To genuinely tackle the scalability problem, a distributed solution is needed that will spread the testbed functionality among multiple entities. These *super nodes* have to be able to interface with the previously described USB infrastructure. In addition, they have to support a secondary communication technology that does not have the size and cable length limits of the USB standard, and forms the *testbed backbone* to which the server and control stations can be attached. Finally, adequate computational, memory and energy resources are needed.

When speaking about indoor deployments, the above requirements can be easily matched using general-purpose PC desktops that have a USB host controller and Ethernet or Wireless Local Area Network (WLAN) network interface. While technologically viable, the solution is still not scalable because of the high costs. Optimally, a device is needed that can satisfy the requirements, while keeping the expenses for a medium to large-scale testbed to a reasonable level. In this sense, the class of 32-bit embedded devices used for attaching networked storage seems to be a very promising alternative since they offer a very attractive cost/performance ratio. They support the USB standard, since this is the primary interface for attaching external flash and hard disks. They have reasonable computational resources (133-266 MHz CPUs, 8-32 MB RAM) and are specifically optimized for dealing with high packet loads.

At the same time, these devices have similar capabilities as the so-called “high-end wireless sensor nodes” or “microservers” [70], enabling dual use of the super nodes as parts of the testbed and as parts of the SUT, to enable experimentation with tiered topologies like Tenet [160].

6.2.5 Testbed Server and Control Station

The server and the control stations must interact with the super nodes using the testbed backbone, so they have to support the same communication technology. Due to the critical role of the server (it contains the testbed database, provides persistent storage for debug and application data from the SUT, runs the daemons that support the system services in the network, etc.) its hardware resources should be adequately dimensioned to guarantee high levels of availability.

The central element of the server architecture is a testbed database that stores a number of tables including configuration data like the registered nodes (identified by the NodeIDs), the sockets and their geographical positions (identified by the SocketIDs) as well as the dynamic bindings between the SocketIDs and NodeIDs. The database is also used for recording debug and application data from the SUT.

Any normal workstation-class machine that is attached to the testbed backbone can serve as a control station. The software interfacing between the testbed server, the super nodes and the control station can be performed using simple RPC mechanisms or more complex Message-oriented Middleware (MOM) solutions.

6.2.6 Summary

The presented TWIST architecture enables flexible support for a range of testing scenarios and configurations:

- Independent SUT operation, where the testbed infrastructure is used only to disseminate the SUT firmware and collect logs. After the initial installation of the firmware, the power-supply control capability is used to remove the USB power, allowing the SUT to operate on batteries in fully independent mode. At the end of the experiment, the testbed infrastructure can be used to collect eventual data stored at the SUT nodes during the experiment run;

- Controlled SUT operation, where the testbed infrastructure is used not only for firmware distribution, but also for power supply. The power supply control capability can be used to “enforce” changes in the topology and emulate (transient) node failures;

- Heterogeneous configuration, where some super nodes become part of the SUT, transparently using some of the attached sensor nodes as their communication interface. This allows modeling of diverse SUT configurations both in terms of power supply, as well as, in terms of available computational capability;

- Hierarchical configurations, where the Ethernet backbone becomes part of the SUT, allowing experimentation with mixed wireless/wired setups necessary for evaluation of handover mechanisms or gatewaying schemes between WSNs and Transmission Control Protocol (TCP)/Internet Protocol (IP) networks.

6.3 CONET Testbed Federation Platform

Similarly to real deployments, the individual WSN testbeds—like the TWIST instance at TKN—lock the evaluation of the system to one particular environment, complicate the differentiation between the intrinsic properties of the SUT and the effect of the particular features and external influences present at a given testbed site.

Testbed	Platform	Nodes	Physical size	Degree		PL	Cost	PL/Cost	Churn
			[m^2 or m^3]	Min	Max				
Tutornet (16)	Tmote	91	$50 \times 25 \times 10$	10	60	3.12	5.91	1.9	31.37
Wymanpark	Tmote	47	80×10	4	30	3.23	4.62	1.43	8.47
Motelab	Tmote	131	$40 \times 20 \times 15$	9	63	3.05	5.53	1.81	4.24
Kanseia	TelosB	310	40×20	214	305	1.45	-	-	4.34
Mirage	Mica2dot	35	50×20	9	32	2.92	3.83	1.31	2.05
NetEye	Tmote	125	6×4	114	120	1.34	1.4	1.04	1.94
Mirage	MicaZ	86	50×20	20	65	1.7	1.85	1.09	1.92
Quanto	Epic-Quanto	49	35×30	8	47	2.93	3.35	1.14	1.11
Twist	Tmote	100	$30 \times 13 \times 17$	38	81	1.69	2.01	1.19	1.01
Twist	eyesIFXv2	102	$30 \times 13 \times 17$	22	100	2.58	2.64	1.02	0.69
Vinelab	Tmote	48	60×30	6	23	2.79	3.49	1.25	0.63
Tutornet (26)	Tmote	91	$50 \times 25 \times 10$	14	72	2.02	2.07	1.02	0.04
Blazeb	Blaze	20	30×30	9	19	1.3	-	-	-

Table 6.1: CTP evaluation on multiple testbeds published in [71].

.

For example, Table 6.1 [71], shows a summary of an experimental evaluation of the functional properties of CTP on the TKN instance of TWIST and eleven additional testbeds. The results illustrate how important protocol performance parameters like number of transmissions per successful delivery (Cost), average path length in hops (PL), or parent change rate (Churn) are influenced by the different testing environments. They suggest that the good performance of TinyCOPS when using CTP as notification dissemination protocol (Section 5.6.2) is partially due to the relatively mild RF interference conditions in TWIST, as compared to other testbeds like Motelab [211].

The variability in the results underlines the need to cross-validate the performance testing of WSN systems under different testbeds as a way of decoupling the influence of the testing environment from the intrinsic properties of the SUT. Unfortunately, the realization of such measurement campaigns is currently accompanied by significant overheads in configuring the experiments and collecting the results on the individual testbeds, since easy experiment migration is hindered by a lack of common management, experiment specification and control infrastructure.

The goal of a testbed federation platform is to address some of these roadblocks by developing a software platform that will enable convenient access to the experimental resources of multiple testbeds organized in a federation of autonomous entities. To this end, we have designed CONET Testbed Federation (CTF), a testbed federation platform customized to the specific needs of the WSN technology. In the following we describe the design principles behind the platform, the core functional decomposition and service APIs, as well as their realization following the Representational State

Transfer (REST) architectural pattern.

6.3.1 Design Principles

Our platform for federating WSN testbeds is built on top of a set of common abstractions for authentication and authorization; resource discovery and reservation; and experiment specification and control. The integration APIs are light-weight, scalable and extensible and aim to preserve high levels of autonomy for the member testbeds in the federation. Due to the specific nature of the wireless medium, we are primarily focused on facilitating experiment migration across the federation members, and not in combining the federation resources into a single "virtual" testbed.

In the design of the platform we have followed a set of guiding principles:

Specialization Although it shares some common characteristic with other testbed federation frameworks, the proposed CTF platform is carefully tuned to the specific needs of the WSN testbeds. This specialization has enabled us to leave out some complex features that are not applicable or not important for the target domain, at the same time giving us opportunity to focus on the more important aspects like the impact of the wireless communication on the resource allocation problem, or the inclusion of tester controlled SUT mobility.

User-centricity In contrast to many other testbed federation frameworks that take an institution centric approach, our platform puts the individual user in the center of the design. It decouples service levels and policies from the question whether a particular user belongs to a federation member institution or not. All aspects of the service should be configurable and controllable at the granularity of a single user. This design principle, for example, has direct implications on the architecture of the AAA abstractions and has significant impact on the openness of the platform.

Scalability A typical WSN testbed is a large heterogeneous distributed system which makes the task of integrating several such systems behind a common federation platform particularly challenging. Addressing these challenges requires careful engineering that leverages the best-practices learned from other successful large distributed systems, like the Web. High scalability of the solution has to be maintained by promoting stateless interactions and using caching whenever possible.

Extensibility To be successful, the CTF platform has to be kept as simple as possible, but also modular and extensible, so new features and capabilities can be organically added when they are needed. In contrast to many other testbed federation frameworks, an explicit goal of the CTF platform is the openness towards external service providers. The same APIs that are used internally to build the higher-level federation services will be also made available to external entities to promote integration with their services.

Coexistence By focusing on a set of common, testbed-independent APIs, the CTF platform will necessarily lack some specific services of the individual testbeds that are valuable to the end-users. Thus, it is crucial to enable parallel use of the native interfaces of the member testbeds and the federation APIs. The CTF platform should not limit the autonomy of the members of the federation by imposing only a single access-path to the resources of the individual testbeds.

Flexibility The usage of the CTF platform should not impose unnecessary constraints on the development process on the user side. In particular, the platform should be accessible using various programming languages and the users should have freedom in selecting the level of abstraction overhead. This should be achieved by offering a basic, language-agnostic set of API, that enables building client-side solutions for raising the level of abstraction.

In the rest of the chapter we discuss how these high level principles have been converted into concrete requirements for the overall architecture of the CTF platform and the features of the core abstractions.

6.3.2 Functional Decomposition

The primary goal of the CTF is to enable convenient access to the resources of multiple WSN testbeds, when organized in a loose federation of testbeds. To illustrate the main architectural features and to establish some common vocabulary, we revisit the use case of performing a cross-validation experimental study on a large number of WSN testbeds.

Figure 6.3 depicts our baseline scenario S_0, reflecting the existing service level. The user U wants to perform a study comprised of a set of experiments $E_1, E_2, \ldots, E_N$ that need to be executed on a set of WSN testbeds: $T_1, T_2, \ldots, T_N$. Each testbed provides a set of SUT resources $R_1, R_2, \ldots, R_N$ necessary for the corresponding experiment.

As it can be seen, without a federation substrate, the user can access the resources of the individual testbeds only over their native APIs: T_i API . This means that for each experiment, and after completing the native authenticating and authorization process, she needs to use a proprietary way to discover and reserve the required resources, to define and control the experiment, and finally, to collect the results.

The user needs to do this on each testbed separately. For N testbeds, she has to potentially use N different interfaces and processes. Without a federation substrate, there is no way to reuse the authentication and authorization credentials, no way to perform resource discovery across the multiple testbeds, no way to reuse the experiment specification, no way to reuse the client-side code for the on-line control of the experiment or for storing and post processing the results. In addition, there is no way to share this content with other users, so they can repeat the experiment and validate the results. There is also no common way to provide access to the results to external service providers that can provide storage or post processing (statistical analysis, plotting, etc.)

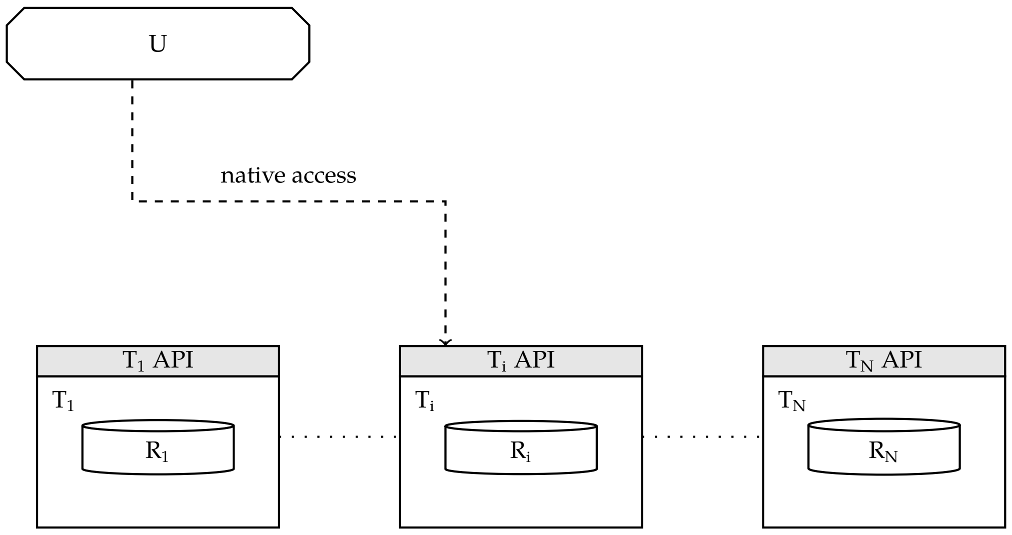


Figure 6.3: Baseline scenario S_0. The user can access the resources of the individual testbeds only over their native interfaces.

The CTF platform tries to address these limitations. Due to the large variability in the services offered by the individual testbeds this requires a two-step approach as described in the following.

As a first step, a common abstraction over the existing capabilities of the testbeds needs to be defined. This abstraction exports an interface, called *Testbed Adaptation API* (TA API), that can be used by the users to access the resources on the individual testbeds via a standardized interface. In this process of interface unification, some specific features of the individual testbeds will undoubtedly remain unrepresented. Due to this, as well as to our coexistence and autonomy principles, we expect that this common interface will be used in parallel, and not instead of the native interfaces I_{T_i}.

The TA API offers a new service level to the users, S_1, leading to the scenario depicted on Figure 6.4. Instead of using the various native testbed interfaces I_{T_i} as in the baseline scenario, users can now access the resources over a standardized API. When they need some testbed-specific capabilities they can always use the native interfaces. By incorporating adequate arbitration mechanisms, the implementation of the TA API will make sure that no conflicts in the access of the resources between the native and the federation users can occur.

Although this service level provides a significant convenience gain with respect to the S_0 scenario, by using only the Testbed Adaptation API (TA API) we are still short of providing some very useful services that abstract over the individual testbeds and operate in the context of the global federation aggregate. For example, we can provide a central place to store the definitions of the experiments, introduce a centralized discovery and reservation system, a central repository of traces and results, etc. For this we need an additional entity, a central *CONET Testbed Federation Server* (CTFS)

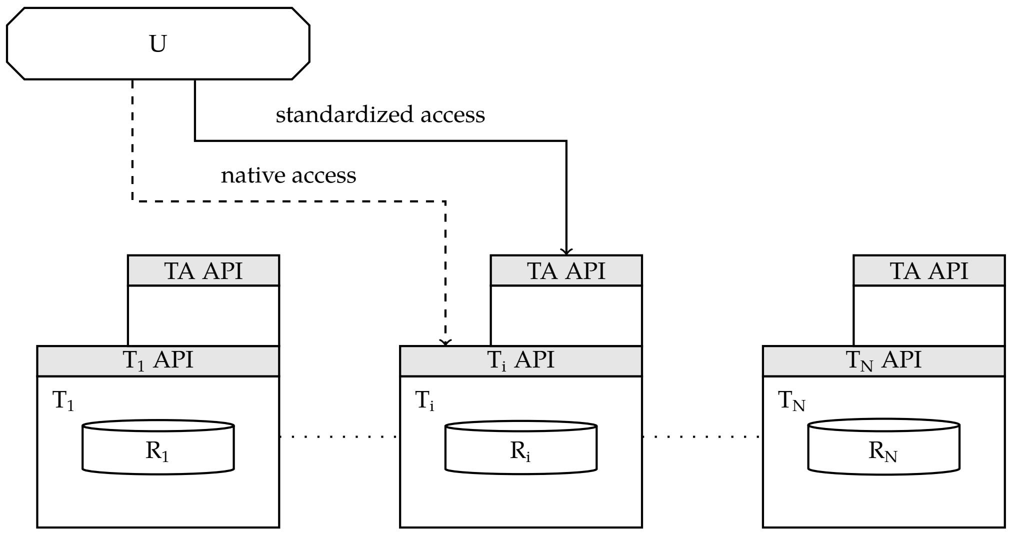


Figure 6.4: Scenario S_1. The Testbed Adaptation API enables standardized access to the resources on the individual testbeds in addition to the native interfaces.

that will offer a second interface class, the *Testbed Federation API* (TF API).

The CTFS leverages the services of the individual testbeds, exported through the standardized TA API, to build a higher level abstraction over the federation aggregate, thus offering new service level to the user, S_2.

Figure 6.5 illustrates the relation between the individual testbed servers exporting the native and adaptation interfaces, the federation server and the users (both native and federation) plus external services in the S_2 scenario. The presented architecture has the flexibility to satisfy the major requirements outlined in our motivating use-case, and represents the basis for the CTF platform solution.

6.3.3 RESTful Implementation

The overall architecture of the CTF platform can be realized using many different architectural patterns that support rich interaction between a set of distributed components. Taking into consideration the main design principles presented in Section 6.3.1, we have decided to base the CTF implementation on the *Representational State Transfer* (REST) architectural style [53].

REST is a set of design constraints for developing rich resource-oriented systems that mirror the scalability and the flexibility of the Web. In the following we provide a short overview of the core properties of systems following the REST architectural style—so called *RESTful systems*–and their benefits in the context of the CTF platform.

Resources

In contrast to Service Oriented Architecture (SOA) and other RPC-based architectural styles where the data is kept private, encapsulated and hidden behind the processing

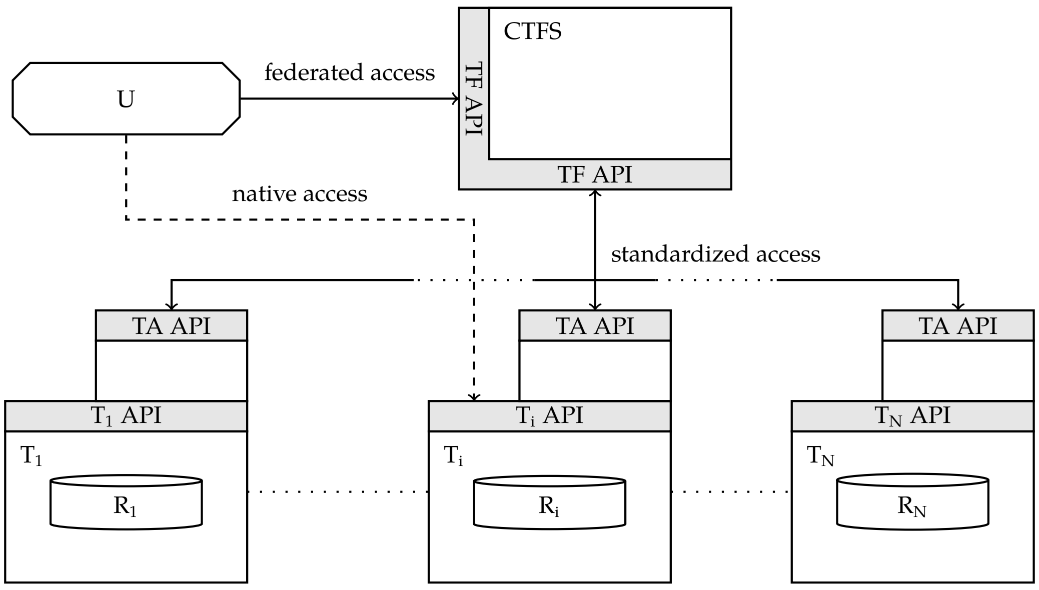


Figure 6.5: Scenario S_2. The CONET Testbed Federation Server leverages the services of the individual testbeds, exported through the standardized Testbed Adaptation API, to build a higher level abstraction over the federation aggregate. The new services are made available to the user over the Testbed Federation API.

components, in REST, the state and the nature of the data elements play a central role.

The *resource* is the main abstraction of information in a RESTful system. The *resource representation* captures the current or intended state of the resource. The components (clients, servers, etc.) act on the resources by transferring and modifying their representations. A *resource identifier* is used to uniquely identify the resource involved in the interaction.

Following the principles of the REST architectural style, we have implemented the two interfaces of the CTF platform as a collection of interconnected resources. Figure 6.6 illustrates the resource model for the Testbed Federation API (TF API). The graph uses the "crow's foot" notation to depict the cardinality of the relations between the individual resources.

Resource Representations

Resources are abstract entities that can only be manipulated through their representations. A resource representation is a sequence of bytes, accompanied by representation metadata that describes those bytes. The components in a RESTful system use *media types* to differentiate between the different possible representations of a resource. One scheme for specifying the media types in the system are Multipurpose Internet Mail Extensions (MIME) types [60].

The agreement on a set of common media types, together with the remaining architectural constraints, give messages in a RESTful system a "self-describing" prop-

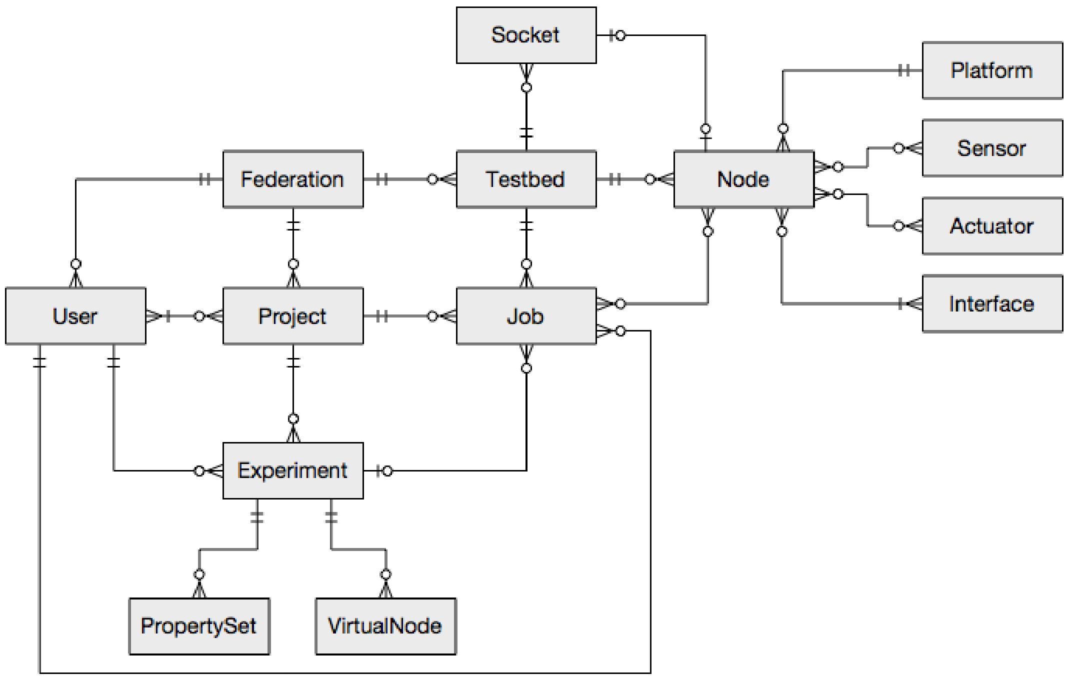


Figure 6.6: Resource graph implementing the TF API.

erty. Using only the media type as indication, the components in the system can safely perform many useful operations without having to look into the body of the message.

The resource model and the definition of the media type(s), used for representing the resources and driving resource and application state, forms the main cognitive effort in designing a RESTful system. Since all components in the system have to agree on the media types used for representing the resources, there is clear benefit in reusing a well defined media type from the MIME register whenever possible. Unfortunately, due to the specifics of the problem that the CTF platform is addressing, we were not able to follow this approach. Instead, we have decided to use a custom media types encoded with in a standard data interchange format.

In contrast to the existing testbed federation frameworks that use documents based on XML (WISEBED uses WiseML, ProtoGENI uses Rspec, etc.), we have opted for JavaScript Object Notation (JSON) [35] as the serialization method for our resources.

JSON is more light-weight and readable then the equivalent XML serialization and is especially suitable for exchanging general data structures. Code for parsing JSON-encoded data exists for large number of programming languages. The use of JSON is also very convenient when developing rich in-browser client side applications for interacting with the CTF platform. Because all JSON text is legal JavaScript code, it is very easy for a JavaScript program to convert the serialized data into an active object. Instead of using a heavy-weight parser, like in the case of XML, one can just use the built-in `eval()` function, passing the JSON encoded representation (after a security check) as parameter. Figure 6.7 illustrates the serialization of the *Node* resource.

```
{
  "uri":"http://cotefe.net/testbeds/123/nodes/456",
  "name":"2343-4A23-4KJ8-5FR5",
  "media":"application/ctf.Node+json",
  "socket":"http://cotefe.net/testbeds/123/sockets/671",
  "platform":"http://cotefe.net/platforms/894",
  "sensors":"http://cotefe.net/testbed/123/nodes/456/sensors/",
  "actuators":"http://cotefe.net/testbed/123/nodes/456/actuators/",
  "power":"true",
  "image":"http://cotefe.net/users/789/images/274"
}
```

Figure 6.7: JSON serialization of the Node *resource.*

The default media type for JSON-encoded representations is `application/json`. Following the approach used in the *Sun Cloud API* [110], for our custom JSON-based representations we use the media type designation `application/ctf.{resource}+json`, where `{resource}` stands for the name of the particular resource that is being represented.

Resource Identifiers

This ability to uniquely identify any resource in the system is a crucial characteristic of RESTful systems and represents the basis for their openness and composability. Although REST does not impose a specific naming scheme for the resource identifiers, in practice they typically take the form of Universal Resource Identifiers (URIs), as defined in the RFC 3986 [13].

There are two major subtypes of URIs: those that also include the "location" of the resource, so that the URI can be immediately dereferenced to get the representation of the resource (i.e. Universal Resource Locators (URLs)) and those that only provide a unique name (Universal Resource Names (URNs) or other Universally Unique Identifier (UUID)-based solutions) without the location part.

The URN identifiers are more stable (for example, a URL can be rendered invalid if the domain name of the server exporting the resources is changed). Many existing testbed federation frameworks like ProtoGENI [169] and WISEBED [218], use URNs as their default identification mechanism. The URNs stability, however, comes at a cost of increased administration overheads and loss of flexibility. The URN namespaces typically need to be registered with an external register [38] like IANA [94] and one loses the standardized dereferencing capability of URL.

We believe that the benefits that URLs carry as common addressing and naming scheme outweigh their shortcomings. The CTF platform uses URLs as resource identifiers exclusively. Any problems associated with non-persistent URLs on the individual testbeds will be handled using simple URL rewriting and redirecting rules in the federation components.

Uniform interface

REST mandates a *uniform interface* between the components in the system. This is the most fundamental differentiation from the other networking styles. In contrast to the rich "verb" space in SOA and other RPC architectural styles, where each object can export different set of methods that operate on its state, in RESTful systems all resources are manipulated with exactly the same method set. In a similar way like the use of standard media types, the use of a generic interface constraints the design freedom but brings significant benefits in return.

Since all resources in the system can be manipulated with the same method set, the components don't have to implement specialized code for accessing different resources in the system. Also, the semantics of each operation is well defined and uniform across all resources and components. This leads to an interface that is easy to understand and simplifies the interoperability between large number of uncoordinated actors. The uniform method set also opens the possibility of using a standardized set of return values to inform the caller about the success of the method invocation.

In RESTful systems implemented using Web technologies, the standard Hypertext Transport Protocol (HTTP) method set serves the role of a uniform interface. The HTTP 1.1 standard [54] defines eight methods: OPTIONS, GET, HEAD, POST, PUT, DELETE, TRACE and CONNECT, out of which the subset: GET, PUT, DELETE and POST are most commonly used in practice. Correspondingly, the HTTP status codes serve the role of return codes.

Table 6.2 provides a succinct overview of the semantics of the most common HTTP methods when applied in a RESTful context.

HTTP Method	Properties	Semantics
GET	Safe, Idempotent	Retrieve a representation of the resource identified by the Request-URI without client-relevant side effects
HEAD	Safe, Idempotent	Same as GET, but only retrieve the header information without the message-body
PUT	Idempotent	Update the resource identified by the Request-URI with the representation contained in the message-body
DELETE	Idempotent	Delete the resource identified by the Request-URI
POST	Not safe, Not idempotent	Accept the representation in the message-body as a new subordinate of the resource identified by the Request-URI; Create a new resource without knowing the final URI; Append to the state of the resource identified by the Request-URI;

Table 6.2: Uniform interface formed by the HTTP method set.

The “Property” column in the table illustrates how having a common set of methods with fixed semantics that is applicable to all resources in the system also leads to a more scalable and robust system.

Issuing a HTTP GET fetches the representation of the resource identified by the URI without any “strings attached”. Just like HEAD, this method is *safe*. It does not result in any change of client-relevant state at the server. Due to this properties, GET can support very efficient and sophisticated caching schemes. At the same time, the GET method is *idempotent*. The client is allowed to issue the same call one or ten times, if he wishes to do so, and the effects to the state of the identified resource at the server is guaranteed to be the same as if he made the call only once. The same idempotent property is shared by the PUT and the DELETE methods. If the client attempts to create or delete a resource and does not receive a positive status code, she knows that she can simply reissue the same request without relevant side-effects.

The HTTP POST method is neither safe nor idempotent. Making two POST requests to a collection resource will likely result in creation of two separate subordinate resources in the collection. The same applies when POST is used in a non-RESTful manner as a way to tunnel data to an arbitrary data-handling process. Because of this, the use of the POST requests has to be limited only to those scenarios where the safe or idempotent methods are not sufficient. In many cases a POST request can be avoided by reorganization of the resource model or by introducing a new resource representing the “result” of the indented activity of the original POST request.

The CTF platform uses the reduced HTTP method set described in Table 6.2 as its uniform interface. Apart from the scalability and robustness benefits explained previously, this also allows clients to cleanly separate the code that deals in generic way with the issuing of the request and the interpretation of the status codes with the service specific aspects that are part of the handling of the resource representations.

For example, fetching information about a particular node in a testbed would be implemented by a GET request on a node resource URI. The same code that handles the GET request can also be used for fetching information on a given platform, only the target will now be a platform resource URI. Similarly, the code that handles the PUT request for creating a new node resource at a given node URI can be shared with the code for creating a new binary image at a given image resource URI.

This should be compared with classical SOA and RPC styles where there are multiplicity of methods taking the role of the uniform method set. For example, the WISEBED API has methods like `getNodeList`, `getCapabilities`, `flashImagesToSensorNodes`, etc. Every service effectively speaks a different language and both the client and the intermediary components (caches, proxies, etc.) can not make any generic assumptions about their semantic properties.

Statelessness

All REST interactions are *stateless*. In Web-based RESTful systems this means that each HTTP request should happen in full isolation and the client has to provide all of the information necessary for the server to understand the request, independent of any requests that may have preceded it.

This constraint, effectively moves the burden for maintaining the *application state* from the server to the client. The *resource state* is maintained on the server and it is the same for every client. If one client changes a particular resource state, this change is made visible to all other clients. It is the responsibility of the client to maintain any state that is specific only to that client.

The statelessness constraint frees the server from the need to retain application state between the individual requests, improving the scalability of the system. It enables parallel processing of the requests without further coordination apart from the resource state. It also allows intermediaries to view each request in isolation. For example, a cache server can make a decision whether to cache or not a result from a request without fearing that state from some previous requests might affect its validity.

Connectedness

Resource representations in RESTful systems contain links to other related resources, allowing the client to *navigate* to them instead of using out-of-band information about the right URI where a given resource representation can be accessed.

A client should be able to effectively use a RESTful APIs without any prior knowledge beyond the initial URI and understanding of the standard media types appropriate for the specific application domain. From the initial URI, all application state transitions must be initiated by the client by selecting among a set of valid next states which are provided by the server as part of the representations of the manipulated resources.

Following this REST constraint, all CTF resources are interlinked (Figure 6.6) and the servers guide the clients through the application state. The testbed servers remain in full control of their URI namespaces. This allows unconstrained evolution of the server-side services by limiting the effects that changes have on client-side code.

The connectedness principle is the main reason why the CTF platform is defined through its resource model and not through the URI space as usual in traditional non-RESTful Web services. As long as the client understands the media types used for representing the CTF resources, they can effectively access all services without prior knowledge of the URI hierarchy.

API Invocation Example

In Figure 6.8 we illustrate a (simplified) invocation of the TF API, intended to register a new experiment on the testbed. The client initiates the process by issuing a POST request to the CONET Testbed Federation Server (CTFS) host *cotefe.net*, using the identifier of the experiments collection resource, associated with the project number 456, */projects/456/experiments* as target. The content of the request is a JSON-serialized representation of an skeleton Experiment resource that contains essential data about like the name of the experiment, the user, associated project, publicity level, etc. Upon successful processing of the request, the server replies with the standard HTTP code “201 Created”, and provides the identifier of the newly created resource *http://cotefe.net/projects/456/experiments/789*.

```
POST /projects/456/experiments
Host: cotefe.net
Accept: application/ctf.Experiment+json
Content-Type: application/ctf.Experiment+json
```

```
{
   "name":"Test␣Experiment",
   "description":"A␣simple␣test␣experiment",
   "user":{
      "uri":"http://cotefe.net/users/123",
      "name":"John␣Smith",
      "media_type":"application/ctf.User+json"
   },
   "project":{
   "uri":"http://cotefe.net/projects/456",
      "name":"Test␣Project",
      "media_type":"application/ctf.Project+json"
   },
   "sharing":"public"
}
```

```
HTTP/1.1 201 Created
Location: http://cotefe.net/projects/456/experiments/789
```

Figure 6.8: Simplified example of registering a new experiment on the CTF using the TF API.

In most cases, this exchange between the clients and the CTFS will be hidden behind convenient programming libraries (for programmatic experiment specification and control) or behind a rich Web client interface (for interactive experiment specification and control).

6.3.4 Summary

We have presented the general design and the RESTful implementation of the CONET Testbed Federation (CTF) platform. We are currently in the process of implementing the first prototypes for the core interfaces. We are developing TA API implementations for TWIST and for several other Cooperating Objects Network of Excellence (CONET) testbeds. For the implementation of the TF API, we have selected the Google App Engine (GAE) framework [180]. The platform as a service model provides us with the required reliability and flexibility and allows us to transparently scale the implementation as new testbeds and users are added to the federation substrate. The proper evaluation of the services of the CTF platform using these prototypes remains important aspect of our future work.

6.4 Evaluation

In this section we evaluate the TWIST testing architecture using its instance in the office building of our research group on the TU Berlin campus. The evaluation of TinyCOPS, the prototype implementation of the DASA interoperability anchor (Section 5.6.2), already demonstrated how the experimental capabilities of the testbed can be effectively applied towards testing the functional and non-functional properties of large-scale WSN systems. In the following, we describe in detail the used infrastructure and provide analysis of the operational data collected during the operation of the testbed in the last three years.

6.4.1 TWIST instance at TKN

Our local instance spans three floors of the Telecommunication Networks Group (TKN) office building at TU Berlin's campus. The testbed is currently equipped with 204 SUT nodes (102 telosb and 102 eyesIFX nodes), making it one of the largest academic WSN testbeds.

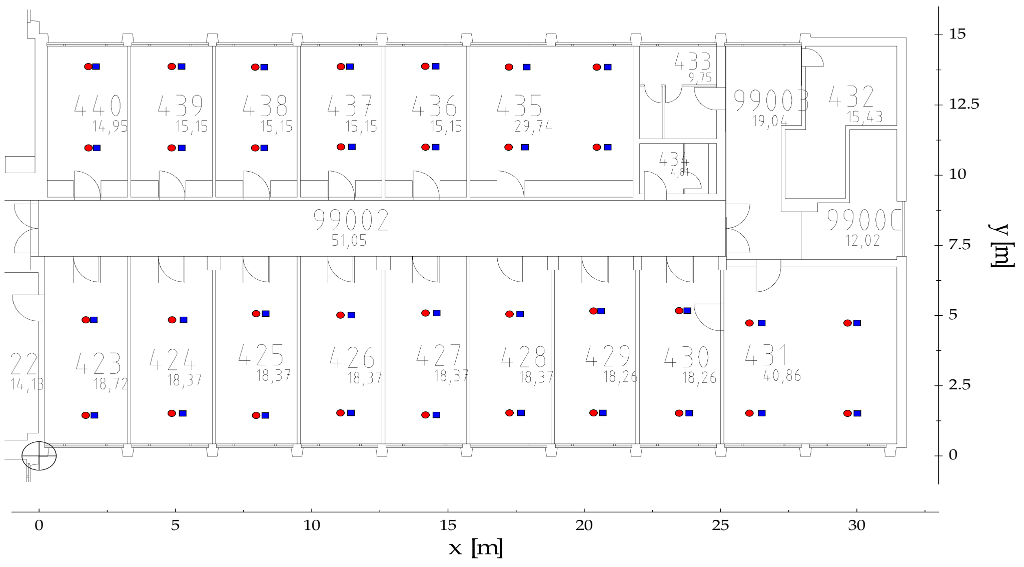


Figure 6.9: TWIST node deployment pattern on the 4th floor of the TKN office building. The squares indicate the locations of the telosb nodes and the circles the locations of the eyesIFX nodes. The same pattern is mirrored on the remaining two floors.

The placement of the nodes is kept as regular as possible, with four to eight sockets in each room, depending on its size. This provides sufficient density for capturing the coarse spatial distribution of measured parameters like light and temperature inside each room, and allows cross-calibration of sensors of the same type. The node deployment pattern for the two SUT platforms is shown in Figure 6.9. Sparser placement patterns can be obtained using the power supply control feature of the testbed architecture (Section 6.2.3).

Hardware Infrastructure

Figure 6.10 schematically depicts the network topology of the testbed backbone infrastructure. Behind the outer firewall, the super nodes and the testbed server are located in a protected network segment allowing increased security and easier service invocation. The web server is isolated in a perimeter segment providing constrained remote access to the testbed services.

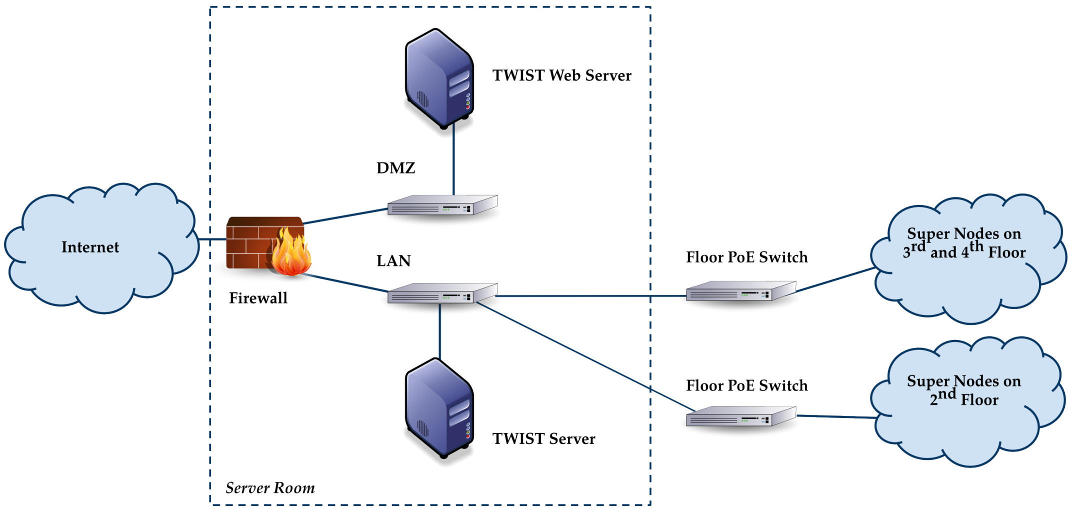


Figure 6.10: Network topology of the TWIST instance in the TKN office building.

Out local instance of TWIST leverages the Network Storage Link for USB 2.0 Disk Drives (NSLU2) device from Linksys (depicted on Figure 6.2(b)) as super node platform. The NSLU2 has two USB 2.0 ports, uses an IXP420 processor from Intel's XScale family (clocked at 133 MHz), has 32 MB of SDRAM and 8 MB of flash as persistent storage. One particular feature of the IXP4xx family are the two integrated "Network Processor Engines" that implement, among else, two full Ethernet MAC and physical layer units along with the related packet-processing functionality. The NSLU2 devices are connected via wired Ethernet to a set of Power over Ethernet (PoE) switches which allow programmatic power supply control of the super nodes in the same fashion as the USB hubs allow programmatic control of the power supply of the WSN nodes.

Our current deployment employs 46 NSLU2 and 50 USB hubs which interface with the testbed sockets using 1335 m-long USB cabling (515 m passive cables and 820 m in active cables). The cables are routed using cable channels mounted along the walls and on the ceiling. To connect a node to a socket, it is simply connected to the USB cable and affixed to the cable channel as shown in Figure 6.2(a). As a result, the nodes are mounted about 4 cm below the ceiling.

For our server infrastructure, we are currently using two Intel Pentium 4 machines with 3.2 GHz CPU, 2 GB of RAM as well as 4x200 GB of hard disk space, organized as RAID 10 for maximal availability.

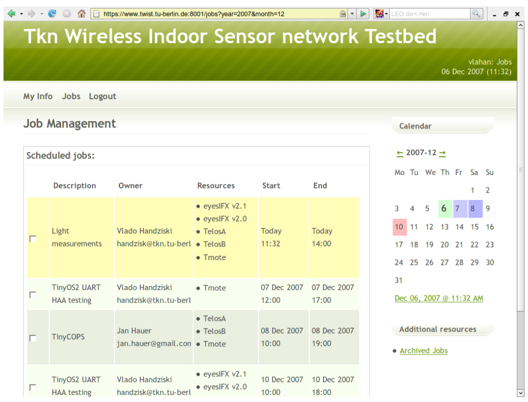
Software Infrastructure

To facilitate their use as super nodes, we have replaced the Linksys-supplied firmware on the NSLU2 platform with a customized OpenSlug [155] distribution of Linux. OpenSlug is a variant of the OpenEmbedded [123] source distribution of Linux that is specially adapted for use on embedded devices.

The firmware customization is motivated by the need to change the default "self-healing" behavior of the Linux kernel when dealing with errors in the communication between the USB host controller and attached USB hubs. We rely on the user-space *libusb* library for injecting control messages that trigger the explained port power-control functions. This library, however, bypasses the USB hub drivers in the Linux kernel. When there are hubs downstream from the powered-off port (i.e. we are controlling the power of a sensor node that is connected with an active cable), this will cause loss of the keep-alive messages between the host controller and the USB hub, which in turn causes the kernel to reset and subsequently re-power the whole USB subsystem, including the sensor node that we wanted to turn-off.

The operating software support for the web and testbed servers is based on Linux: a vanilla CentOS 5 distribution on the web server, and a Debian Etch distribution on the testbed server. The central testbed database is implemented using PostgreSQL. The root and swap file systems of the super nodes are centrally hosted on the testbed server and served over Network File System (NFS). For the management of the super node network, the testbed server also runs all the necessary system services like Dynamic Host Configuration Protocol (DHCP), Domain Network System (DNS), Network Time Protocol (NTP), etc.

On each super node, a number of Python scripts are deployed, supporting the basic testbed functionality like sensor node programming, executing power supply control, injecting and collecting data, and more. The invocation of these actions is controlled by a daemons resident on each super node that receive commands from the testbed server using RPC.


https://www.twist.tu-berlin.de:8001/jobs?year=2007&month=12

Tkn Wireless Indoor Sensor network Testbed

vlahan: Jobs
06 Dec 2007 (11:32)

My Info Jobs Logout

Job Management

Scheduled jobs:

	Description	Owner	Resources	Start	End
☐	Light measurements	Vlado Handziski handzisk@tkn.tu-berl	• eyesIFX v2.1 • eyesIFX v2.0 • TelosA • TelosB • Tmote	Today 11:32	Today 14:00
☐	TinyOS2 UART HAA testing	Vlado Handziski handzisk@tkn.tu-berl	• Tmote	07 Dec 2007 12:00	07 Dec 2007 17:00
☐	TinyCOPS	Jan Hauer jan.hauer@gmail.con	• TelosA • TelosB • Tmote	08 Dec 2007 10:00	08 Dec 2007 19:00
☐	TinyOS2 UART HAA testing	Vlado Handziski handzisk@tkn.tu-berl	• eyesIFX v2.1 • eyesIFX v2.0	10 Dec 2007 10:00	10 Dec 2007 18:00

Calendar

← 2007-12 →

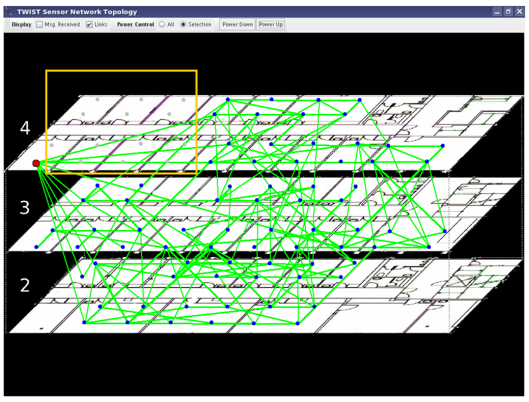
Mo	Tu	We	Th	Fr	Sa	Su
					1	2
3	4	5	6	7	8	9
10	11	12	13	14	15	16
17	18	19	20	21	22	23
24	25	26	27	28	29	30
31						

Dec 06, 2007 @ 11:32 AM

Additional resources

• Archived Jobs

(a) Web interface for job registration and control.



(b) GUI for controlling the power supply to the SUT nodes.

Figure 6.11: TWIST control interface.

All of the available testbed services are exported to the users through a rich web service API, hosted on a dedicated web server, situated in a protected perimeter network segment (Figure 6.10). To guarantee maximal scalability and availability, the web server and the supporting background services are implemented using Twisted [200], a high-performance event-driven networking engine written in Python.

The web service API provides support for job scheduling and job control including firmware programing, power-supply control and automatic tracing. The API can be accessed either interactively, through the testbed web site (Figure 6.11(a)), or through programming libraries like cURL, enabling the development of rich interaction clients like the one depicted in Figure 6.11(b), used for monitoring of the routing topology and interactive node fault injection (using the SUT power supply capability of TWIST). In addition, the testbed users can directly interact with the serial interfaces of the SUT nodes through dedicated Secure Shell (SSH) tunnels.

6.4.2 Power-supply Control

The power-supply control capability using the USB 2.0 standard is one of the main features of the TWIST architecture. Depending on whether the sensor node attached to the socket has a battery or not, the power supply control capability of TWIST enables four different power-supply transitions (Section 6.2.3).

In the following, we evaluate the impact of these commands on the supply voltage of the attached WSN nodes. Figure 6.12 shows the time response of the MCU supply voltage on an eyesIFXv2 and on a telosb node during two such transitions, captured using a high-speed digitizer. The nodes are connected to a Linksys USB2HUB4 hub (shown in Figure 6.2(b)).

The results confirm that the transitions are tightly time bounded, with the MCU voltage dropping below the brown-out protection point of 1.8 V within 20 ms of the hub receiving the power-off command. The same tight bounds are observed for powering-on as well as for the transitions between USB and battery-powered states. The small differences in the response between the two platforms are due to different designs of the respective power-supply subsystems. Furthermore, our evaluation of the USB cabling has shown that up to 10 m the length of the USB cable between the controlling hub and the sensor node has negligible influence on these delays. This quick time response makes the power-control capability one of the most useful features of the TWIST architecture.

It is important to note that in our testing we have detected several hubs on the market that claim to fully support the USB 2.0 standard, but fail to support the port power-control feature. This function is rarely used by “normal” users, and it requires additional power-control circuitry in the hub, so we suspect that many vendors have decided to silently drop this feature. From our extensive testing of different hubs, we can conclude that the problem is present in almost all hubs having a controller chip from Genesys Logic. The ones that fully support the standard tend to have a controller chip produced by NEC. In particular, we can confirm the proper functioning of the

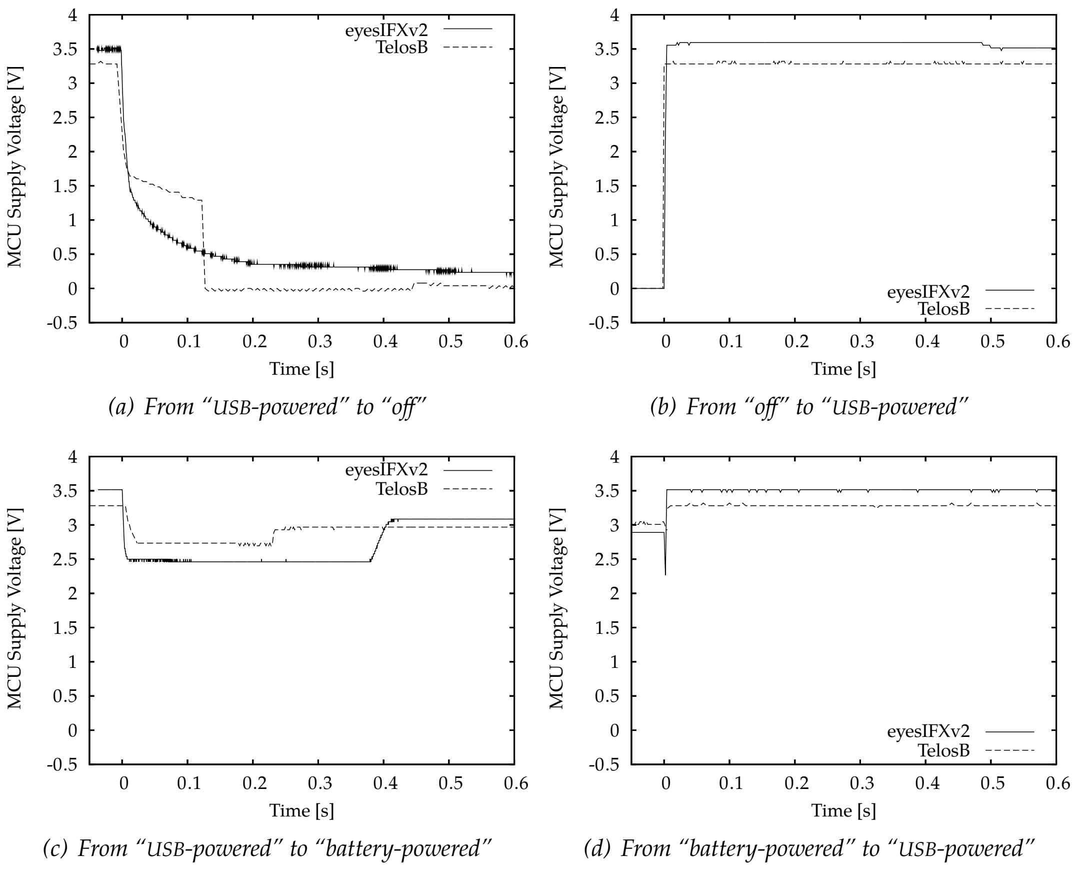


Figure 6.12: Time response of the sensor node MCU supply voltage after the respective USB hub has received a port power-control command from the testbed.

DUB-H4 rev.A1 hub from D-Link (now out of production, the later revisions B1, B2 and B3 silently ignore the port power-control commands) and the USB2HUB4 from Linksys which is used at our local TWIST instance.

6.4.3 Testbed Performance

Our testbed infrastructure is fully instrumented using Simple Network Management Protocol (SNMP). The monitoring data is visualized using Cacti [22], a front-end for the RRDtool [179] for collecting and processing time-series data in Round Robing Database (RRD) format. The constant monitoring of the infrastructure health and the automatic notification of failures has been crucial aspect for maintaining high levels of availability of the testbed resources. A second source of valuable operation data is the testbed database that contains records for the performed experiment jobs since the activation of the testbed. In the following we analyze some of the collected data and use it to illustrate selected aspects of the testbed operation.

Time Synchronization and Availability

The causality analysis among a sequence of distributed events generated by the SUT nodes is a fundamental task in distributed testing. The standard solution involves comparing the event time-stamps and requires synchronization between the SUT nodes where the time-stamps are generated.

The TWIST infrastructure offers a simpler but less precise alternative. When the SUT dumps debug data, this data can be time-stamped by the super nodes. Figure 6.13 shows an excerpt of a the automatic tracing logs generated by TWIST with time-stamped SUT messages using the testing infrastructure. As long as the synchronization of the distributed testbed infrastructure remains better than the event generation period, the relative ordering of the events can be correctly reconstructed from the time-stamps.

```
...
# 1291989432.026970 Connection established to node 196
# 1291989432.027328 Connection established to node 200
# 1291989432.027493 Connection established to node 199
# 1291989432.027661 Connection established to node 202
# 1291989432.027832 Connection established to node 203
...
1291989432.041105 196 00007e00ce163f8100ba00ce000a0100070110000201dd590008000700cb
1291989432.041181 199 00007e00e1163f8100e100e100e0010007000900010 2e1680007000500e8
1291989432.041146 200 00007e00f9163f8100f900f9000a0100080008000101d556000400040101
1291989432.043344 202 00007e0053163f8100530053000a0100080008000501de6900070004005b
1291989432.041216 203 00007e0057163f810057005700cb0000070011000103dc66000a0007005e
...
```

Figure 6.13: *Excerpt of a TWIST trace file, showing testbed-based time-stamping of the SUT debug messages. The time-stamps are stored in UNIX time representation, showing the number of seconds from 01/01/1970.*

We use the Network Time Protocol (NTP) [144] to keep the testbed infrastructure synchronized. To tune the performance of the protocol we have disabled the peer-to-peer mode and rely on periodic synchronization broadcasts from the testbed server that synchronizes the super nodes. Figure 6.14 shows the maximal time synchronization error (maximal offset across all super nodes) in the testbed, collected from the NTP daemons on the super nodes. The graph covers the last two years of operation of the testbed, and each data point represents the maximal time synchronization for a 24-hour period.

The graph shows rare peaks in the synchronization error (bounded below 100 ms), which are associated with disruptions in the testbed infrastructure like replacements of failed super nodes hardware or rebooting of the time synchronization server. The LOESS curve and its 95% confidence interval shows that the average time synchronization error remains lower than 10 ms. The curve also shows visible improvement in the error after 09/2009, when the super nodes were migrated to PoE, indicating that the stability of the power supply has noticeable influence on the quality of the time synchronization. During 2010, the average error remained in the range of 1–2 ms.

The SUT platforms send packets at most every few milliseconds, making the precision of the testbed-based time-stamping sufficient even for high-load scenarios.

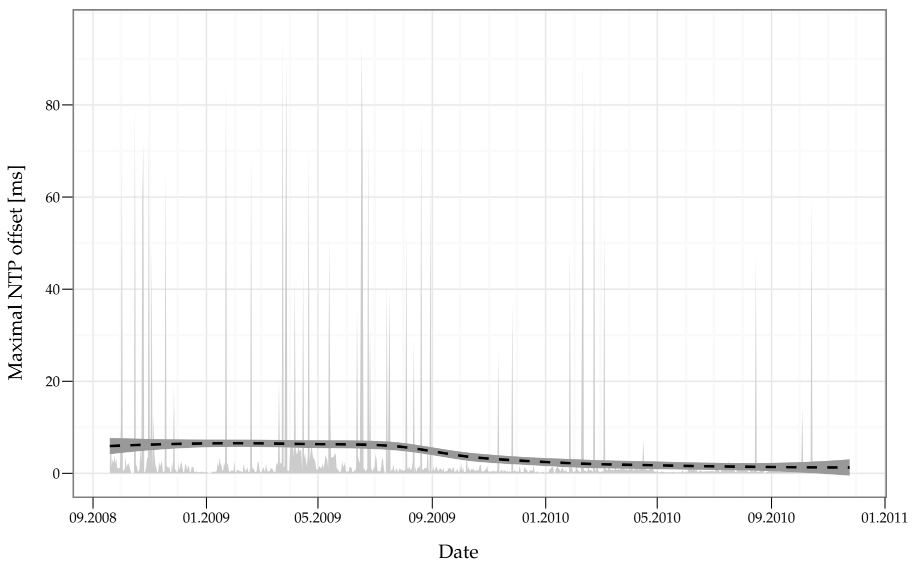


Figure 6.14: Maximal time synchronization error (NTP offset) in the protected network segment containing the super nodes and the sensor testbed.

The super nodes are central element of the TWIST testbed infrastructure and their availability is good indication for the overall health of the testbed infrastructure. Figure 6.15 provides an overview of the super node availability data over the last two years of operation of the testbed. The gray bars show the average number of *accessible* super nodes in a 24-hour period, while the thick black line represents the total number of *deployed* super nodes.

The data confirms the high availability of the testbed infrastructure in the surveyed period. Apart from few maintenance events when all of the super nodes were shortly unavailable, typically we experience about 1–2 super node errors per week, which have to be remedied using the remote power supply control built on top of the PoE infrastructure. The noticeable degradation of the availability in the last three moths of operation is the result of the extensive reconstruction activities in the TKN building which have negatively affected the affect the availability and the stability of the testbed infrastructure.

Usage Patterns

The TKN instance of TWIST operates under a very liberal usage policy that provides full open access to the testbed infrastructure to all academic researchers, while maintaining priority access for the members of the TKN group. In contrast to most other public WSN testbeds, our usage policy does not impose limitations on the duration of the testbed jobs. This fact, combined with the long operational lifetime

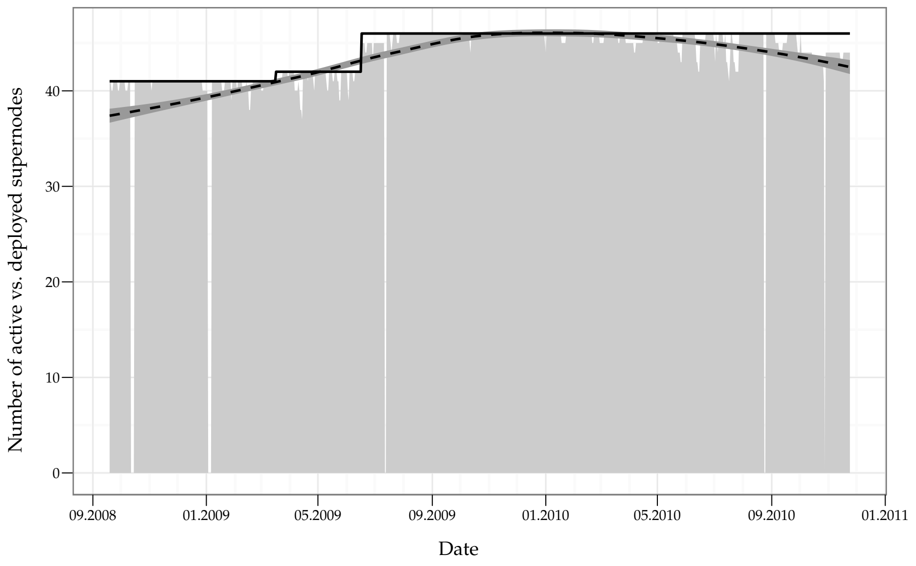


Figure 6.15: Number of accessible super nodes, with respect to the total deployment.

of the testbed, makes the archived records of the testbed jobs, a valuable source of information about the experimental needs of the researchers.

Figure 6.16 shows a histogram of the number of testbed jobs completed per month of operation of the testbed during the last three years. The data is clustered based on the SUT platform used in the experiment. The results show satisfactory utilization of the testbed resources and clear preference for experiments using the tmote sky/telosb platform. The large peak in performed experiments in 05/2008 is a result of an extensive experimental evaluation of the Arbutus data collection protocol [170].

Figure 6.17, shows the empirical Cumulative Distribution Function (CDF) of the experiment duration. From one side, the results show that 50 % of the experiments took less than 3 hours. From the other side, 20 % of the jobs on our testbed lasted more than one day, with the longest recorded job experiment, using the eyesIFX nodes, taking 36 days and 5 hours. The results confirm the benefit of having significantly longer experiment slots than the typical 30 minute to 120 minute limits imposed by the other public WSN testbeds. Of course, the need for longer experiments needs to be checked against the need for fair access. When the utilization of the testbed increases further, more sophisticated scheduling solutions like Mirage [32] might be needed to achieve this balance.

6.5 Related Work

Motelab [211] is a very popular WSN testbed solution. It provides an Ethernet back-channel to each sensor node in the network by attaching a dedicated Stargate board [36]. The interaction between the users and the testbed is batch-oriented and is

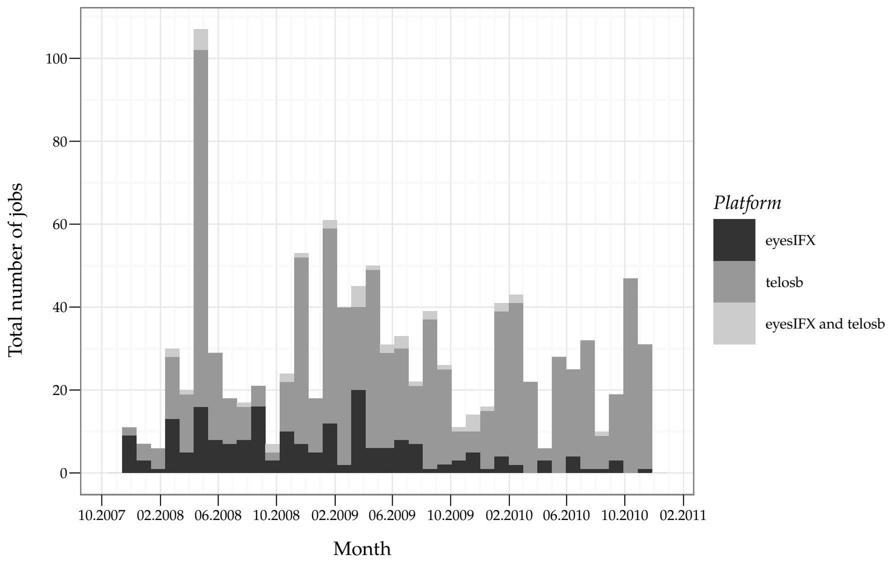


Figure 6.16: Number of testbed jobs performed each month and the used SUT platform.

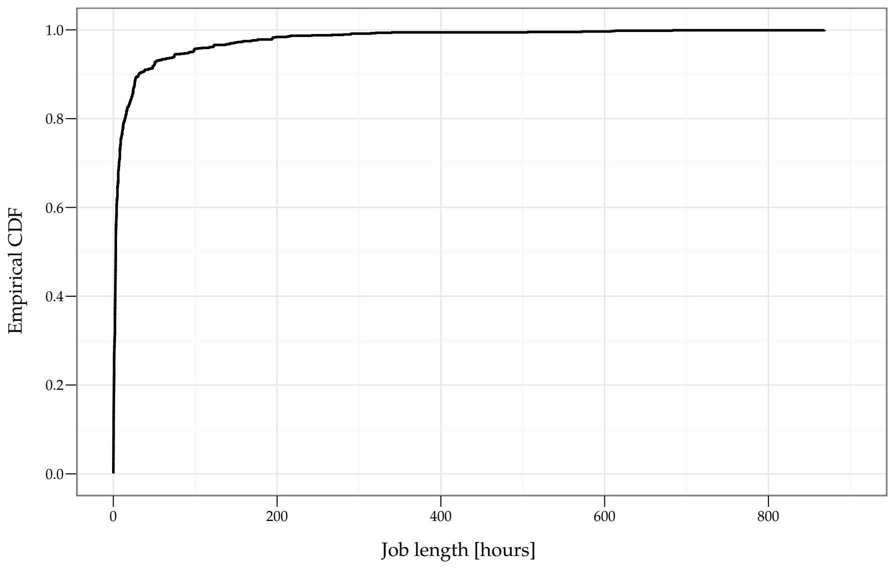


Figure 6.17: Empirical CDF of the duration of the testbed jobs.

controlled via a dynamic web interface supported by a back-end database. Like Motelab, the Kansei testbed [202] also uses Stargate boards, but it allows richer interaction with the SUT. It uses the EmStar development system for Linux-based WSNs [69] and, similarly to TWIST, allows evaluation of both flat and hierarchical WSNs with different communication technologies. The WASAL testbed [216] provides a wired back-channel using Serial-to-Ethernet devices. These devices act only as communication bridges and range extenders, not actively participating in the distributed execution of the testbed functions.

Despite having dedicated, wired back-channels, none of the testbeds listed above offers any power control functions. This capability is one of the main features of TWIST, contributing to its high versatility. Also, they use a one-to-one mapping between the sensor nodes and the "concentrators". This and the custom nature of the "concentrators" make the above solutions much more expensive than TWIST.

The Omega architecture [201] resembles the lower tier of TWIST. Both testbeds rely on the standardized USB interface available on some of the newer mote platforms, and both use USB-hubs to bridge the 5 m length barrier. Nevertheless, the Omega is not utilizing the features of the USB 2.0 standard and is not exporting a power-control function to its users. Its design is also less scalable. The USB-hub daisy-chaining approach taken in Omega can only scale up to 127 USB devices. With the super node tier, TWIST is not suffering from such restrictions.

Platforms with two wireless transceiver can also be used for out-of-band signaling [49]. While more powerful than single channel solutions like [181], this approach shares many of its problems. For example, switching a single node on and off is nontrivial and an emulation of this feature is needed. Doing this simultaneously and reliably for a large number of nodes demands sophisticated protocols to ensure that all targeted nodes have indeed received the command. These protocols must share the scarce node resources with the SUT application and protocol stack code. Additionally, the lack of cabling requires that the node batteries are changed from time to time, creating significant maintenance work for the testbed owner.

6.6 Summary

In this chapter we have introduced the design of TWIST which addresses many of the requirements we have identified for WSN testbeds: support for different application network architectures, control over the network topology, fast reprogramming, distributed debugging and a high degree of scalability.

The feasibility of the TWIST design has been demonstrated by building a large-scale instance in our office building. The TKN instance of TWIST is one of the largest publicly accessible WSN testbeds and is being successfully used by the research community in support of their disseminating activities. TWIST leverages affordable hardware and on open-source software. Our own software components are made available to the community via the testbed website. This makes our architecture a valuable testbed template which has already been instantiated by several external groups [52, 195].

To address the existing roadblocks in performing cross-validation studies on multiple WSN testbeds, we have designed a RESTful platform that enables convenient access to the experimental resources of multiple testbeds, organized in a federation of autonomous entities. The platform is built on top of common abstractions for authentication and authorization; resource discovery and reservation; and experiment specification and control. The federation APIs are light-weight, scalable and extensible, and aim to preserve high levels of autonomy for the member testbeds.

CHAPTER 7

Conclusions

7.1 Double-anchored Software Architecture

The current software development process in WSNs is characterized by many inefficiencies. The application-specific nature and the constrained resources push developers into closed and vertically-integrated solutions that impede design and code reuse and hinder faster growth. Developing applications is hard and requires expertise across the technological stack, starting from the hardware platform, communication protocols, sensing stack, up to the application domain.

In this dissertation we have argued about the benefits from introducing a broad domain-specific software architecture for WSNs that can codify the functional decomposition of the system and lay foundations for a more structured development approach. To this end, we have proposed the Double-Anchored Software Architecture (DASA) for WSN, a novel architectural framework that balances between the need for stable abstractions and code reuse and the need to optimize the system to the specific requirements of the target application and hardware.

DASA is based on careful identification of those parts in the software stack where interface fixation can maximize the reuse gains, and on identification of the parts with decisive impact on the fidelity and the efficiency, which should be kept flexible to enable target-specific optimization. Our architectural framework identifies two anchorage zones where rigidity can contribute towards increasing the decoupling and the reuse in the WSN software development process: a *portability anchor*, that abstracts the local services provided by the underlying hardware; and an *interoperability anchor* that abstracts the services in remote contexts and allows more rapid development of distributed WSN applications.

7.2 Portability Anchor

The lower anchor in DASA addresses the problem of complexity hiding and portability in the hardware abstraction code. Its interfaces shield the rest of the system from the intricacies of low-level hardware access, enabling easy development of portable service and application code. At the same time, the anchor makes platform-specific optimizations possible when the fidelity and efficiency costs of the portability abstractions become unacceptable.

When we started with our work on the portability anchor, all popular WSN execution environments lacked a clear organization of the hardware abstraction code and exported ad-hoc abstraction interfaces that were strongly biased by the features of particular hardware platforms. Through our experience with porting different hardware platforms on these early WSN operating systems, we realized the benefits of separating between the development of complexity hiding abstractions and their equalization across different hardware platforms. This led us to the three-layer hardware abstraction architecture that gives the DASA portability anchor its flexibility.

We applied these insights into the design of TinyOS 2.x, a popular execution environment for WSN, as one of its main design features. Currently, more than ten TinyOS Enhancement Proposal provide detailed specification for service abstractions like timing, communication, sensing, storage, power management, etc., aligned with the decomposition principles of the portability anchor. These specifications are supported by reference implementations that collectively comprise a broad prototype of the proposed anchor architecture. In this work we leveraged the TinyOS 2.x reference implementations to evaluate the proposed portability anchor design. Using a combination of micro-benchmarks and test applications we demonstrated the successful achievement of the design goals and illustrated the flexibility in balancing between the need for reuse and the fidelity costs of the complexity hiding.

The proposed design of the portability anchor has been currently validated only in the specific context of TinyOS. An important direction of future work is to evaluate the possibilities for expanding the architectural principles of the portability anchor across *multiple* WSN execution environments. The WASP project [157] follows a similar goal of establishing an OS-independent abstraction interface. Among other things, this would facilitate the development of hardware and OS-independent protocol stacks, direction promoted by the OpenWSN [158] project.

We see several challenges that have to be met. The differences in the code organization model in the different execution environments represent the first roadblock. DASA assumes a component-based organization, but due to the source-level focus of the specification and the granularity of the design, we believe that the architectural constraints can be easily applied to other domains. More challenging difficulties can arise due to "impedance mismatch" in the API design philosophies. The DASA portability anchor promotes wide and expressive interfaces, supported by extensive compile-time checking which can be hard to reconcile with narrow, POSIX-like designs like the one used in MantisOS. Finally, the differences in the synchronicity of the service invocations would need to be addressed. Unconstrained access to the hardware resources, as promoted by the Hardware Adaptation Layer (HAL) of the DASA portability anchor typically implies asynchronous access. This can be a problem for those WSN execution environments that only support blocking semantics.

7.3 Interoperability Anchor

The interoperability anchor in DASA addresses the problem of providing interoperable access towards the system services both in local and in remote context, as basis for a more rapid development of distributed WSN applications. The anchor exports a customized Content-Based Publish/Subscribe (CBPS) service that is fully decoupled from the underlying services, including the communication stack. This decomposition strives to provide to the application developer simple and flexible means for adapting the service to the specific needs of the target application.

The design of the interoperability anchor is optimized to the specific needs of resource constrained WSN platforms. Consequently, it does not provide rich mechanisms for run-time adaptation like introspection and reflection. Most of the configurability freedom is limited to compile-time. For run-time signaling, DASA relies on explicit exchange of *metadata* among the components in a local execution context, as well as across different WSN nodes and contexts.

To evaluate the design of the DASA interoperability anchor we have developed TinyCOPS, a prototype implementation in the context of the TinyOS execution environment. Using TinyCOPS, we have experimentally demonstrated the flexibility and extensibility of the design and its ability to support different communication protocols and interaction patterns. Our experience with the prototype framework suggests that by careful component decomposition and interface design, it is indeed possible to achieve a good balance between efficient resource usage and reusable software design, which are the core design goals of DASA.

The presented design of the interoperability anchor in DASA can be extended in several directions. One area of interest is the better integration of efficient service discovery mechanism on top of the CBPS service [83, 183]. Another promising avenue of future work is to explore gateway solutions that will enable transparent bridging between the CBPS service of the DASA interoperability anchor and external data-centric frameworks based on publish/subscribe extensions of the Extensible Messaging and Presence Protocol (XMPP) [143], Google's PubSubHubbub protocol [73] or popular MOM solutions like Advanced Message Queuing Protocol (AMQP) [205] and ZeroMQ [223].

7.4 Distributed Testing Infrastructure

DASA promotes a decoupled software development process in which the principles of "black-box" reuse can lead to significant reduction in the development overheads. This compositional freedom, however, has to be supported by stronger integration testing that will validate the semantics of the services and their non-functional properties. This process requires distributed testing infrastructure that will allow realistic but controlled testing at deployment-scales.

To address these needs, we have designed TWIST, a flexible testing infrastructure for WSNs that enables efficient testing of functional and non-functional properties of distributed WSN services. TWIST provides basic services like node configuration, network-wide programming, out-of-band extraction of debug data and gathering of application data. It also offers powerful topology control and fault injection capabilities that we leveraged in our evaluation of the DASA interoperability anchor to illustrate the influence of the underlying communication protocols on the semantics of the exported service.

The TKN instance of TWIST is one of the largest publicly available WSN testbeds and its services have been successfully used by the members of the WSN research community in support of their dissemination activities. Because TWIST relies on affordable hardware and open-source software, it serves as testbed template that has been replicated by several external groups.

To address the existing roadblocks in performing cross-validation studies on multiple WSN testbeds, we have also designed a RESTful platform that enables convenient access to the experimental resources of multiple testbeds, organized in a federation of autonomous entities. With this, we are contributing to the ongoing effort in the community towards federated testbeds solutions, as exemplified by the related activities under the FIRE [56] and GENI [68] initiatives.

Our current work on the testing federation concentrates on refining the platform design and implementing the first prototypes for the core interfaces. As future work we would like to leverage the standardized interfaces of our platform to build template experiments that can serve as a reusable benchmark suite for automatic evaluation of WSN services in context of multiple testbeds. Such suite can reduce the barriers for cross-validation studies and can offer a common base for comparing the functional and non-functional properties of competing designs and implementations.

APPENDIX A

Hardware Platforms Survey

This appendix presents the data from our survey of WSN platforms which we used as input for establishing the main features and trends in WSN hardware design, analyzed in section 2.2. The list of the platforms and their main components was compiled by combining information from a number of primary and secondary sources: [12, 14, 17, 29, 39, 40, 58, 78, 89, 99, 108, 114, 133, 134, 135, 142, 176, 204, 210, 219]. The features of the individual processing elements and transceivers were extracted from the product data sheets.

A.1 Surveyed Platforms

Platform	Processor	Transceiver	Release
Rockwell *AWAIRES1*	Intel *SA-1100*	Conexant *RDSSS9M*	1998
UC Berkeley *WeC*	Atmel *AT90LS8535*	RFM *TR1000*	1998
UC Berkeley *Rene*	Atmel *AT90LS8535*	RFM *TR1000*	1999
Intel *SpotON*	Freescale *MC68EZ328*	RFM *TR1000*	1999
MIT *µAMPS*	Intel *SA-1100*	National *LMX3162*	1999
UC Berkeley *Dot*	Atmel *ATmega163*	RFM *TR1000*	2000
UC Berkeley *Rene2*	Atmel *ATmega163*	RFM *TR1000*	2000
ETH Zurich *BTnode1*	Atmel *ATmega128L*	Ericsson *ROK101008*	2001
UCLA *Medusa MK-2*	Atmel *ATmega128L*	RFM *TR1000*	2001
	Atmel *AT91FR4081*		
Crossbow *Mica*	Atmel *ATmega103*	RFM *TR1000*	2001
U. Colorado *Nymph*	Atmel *ATmega128L*	Chipcon *CC1000*	2001

Table A.1: Overview of the surveyed platforms and their main components [Continued…]

Platform	Processor	Transceiver	Release
UC Berkeley *PicoNode1*	Intel *SA-1100*	Ericsson *ROK101007*	2001
	Xilinx *XC4020XLA*		
ETH Zurich *Smart-ist ETH*	Atmel *ATmega103L*	Ericsson *ROK101007*	2001
Teco *Smart-its*	Microchip *PIC16F876*	RFM *TR1001*	2001
Sensoria *WINS NG 2.0*	Hitachi *SH-4*	Sensoria *WINS NG RF 2.0*	2001
Sensoria *WINS3.0*	Intel *PXA255*	Sensoria *WINS NG RF 2.0*	2001
Rice U. *GNOMES*	Texas Instruments *MSP430F149*	National *LMX9820*	2002
UCLA *iBadge*	Atmel *ATmega128L*	Ericsson *ROK101007*	2002
	Texas Instruments *TMS320VC5416*	RFM *TR1000*	
Crossbow *Mica2*	Atmel *ATmega128L*	Chipcon *CC1000*	2002
Crossbow *Mica2Dot*	Atmel *ATmega128L*	Chipcon *CC1000*	2002
Crossbow *MicaZ*	Atmel *ATmega128L*	Chipcon *CC2420*	2002
UC Berkeley *PicoNode 2*	Intel *SA-1100*	Proxim *RangeLAN2*	2002
MIT *PushPin*	Texas Instruments *C8051F016*	Newark *83F8851*	2002
ScatterWeb *ESB/2*	Texas Instruments *MSP430F149*	RFM *TR1001*	2003
U. Southampton *Glacsweb Probe*	Microchip *PIC18F8722*	Radiometrix *TX1H*	2003
Intel *IMote1*	Zeevo *TC2001P*	Zeevo *TC2001P*	2003
U. Karlsruhe *Particle*	Microchip *PIC18F252*	RFM *TR1001*	2003
MIT *RFRAIN*	Chipcon *CC1010*	Chipcon *CC1010*	2003
UC Berkeley *Spec*	UC Berkeley *Spec*	UC Berkeley *Spec*	2003
U. Tokyo *U3*	Microchip *PIC18F452*	RFM *TR3001*	2003
Bradley U. *Wisenet*	Chipcon *CC1010*	Chipcon *CC1010*	2003
Philips Research *AquisGrain*	Atmel *ATmega128*	Chipcon *CC2420*	2004
UF Minas Gerais *BEAN*	Texas Instruments *MSP430F169*	Chipcon *CC1000*	2004
Imperial CL *BSN node*	Texas Instruments *MSP430F149*	Chipcon *CC2420*	2004
Cork IT *CITNode*	Microchip *PIC16F877*	Nordic *nRF903*	2004
UC Cork *DSYS25*	Atmel *ATmega128L*	Nordic *nRF2401*	2004
Infineon *EyesIFXv1*	Texas Instruments *MSP430F149*	Infineon *TDA5250*	2004
Nedap *EyesNEDAP*	Texas Instruments *MSP430F149*	RFM *TR1001*	2004
Harvard U. *Pluto*	Texas Instruments *MSP430F149*	Chipcon *CC2420*	2004
U. Edinburgh *ProSpeckz*	Cypress *CY8C27643*	Chipcon *CC2420*	2004
UC Berkeley *TelosA*	Texas Instruments *MSP430F149*	Chipcon *CC2420*	2004
UC Berkeley *TelosB*	Texas Instruments *MSP430F149*	Chipcon *CC2420*	2004
Moteiv *Tmote Sky*	Texas Instruments *MSP430F1611*	Chipcon *CC2420*	2004
CSEM *WiseNET*	CSEM *WiseNET*	CSEM *WiseNET*	2004
Princeton U. *ZebraNet*	Texas Instruments *MSP430F149*	MaxStream *9xStream*	2004
Dynastream *Ant*	Texas Instruments *MSP430F1232*	Nordic *nRF2401*	2005
Fraunhofer *AVM Demonstrator*	Texas Instruments *MSP430F149*	Nordic *nRF2401*	2005
CEI-UPM Spain *Cookie*	Analog Devices *ADUC841*	Telegesis *ETRX2*	2005
Ember *Ember*	Atmel *ATmega128L*	Ember *EM250*	2005

Table A.1: Overview of the surveyed platforms and their main components [*Continued…*]

Platform	Processor	Transceiver	Release
EnOcean *EnOcean*	Microchip *PIC18F452*	Infineon *TDA5250*	2005
Infineon *EyesIFXv2*	Texas Instruments *MSP430F1611*	Infineon *TDA5250*	2005
CSIRO *Fleck*	Atmel *ATmega128L*	Nordic *nRF2401*	2005
Intel *IMote2*	Intel *PXA271*	Chipcon *CC2420*	2005
Luela U. *Mulle*	Renesas *M16C/62P*	Mitsumi *WML-C10AHR*	2005
MIT *Parasitic node*	Silicon Labs *C8051F311*	Blueradios *BR-C11A*	2005
U. Karlsruhe *Particle 2/29*	Microchip *PIC18F6720*	RFM *TR1001*	2005
UCLA *RISE*	Chipcon *CC1010*	Chipcon *CC1010*	2005
	Renesas *M16C/28*		
Sensinode *SensiNode*	Texas Instruments *MSP430F1611*	Chipcon *CC2420*	2005
Intel *Shimmer*	Texas Instruments *MSP430F1611*	Chipcon *CC2420*	2005
U. Tokyo *SolarBiscuit*	Microchip *PIC18F452*	Chipcon *CC1020*	2005
Sun Microsystems *SunSPOTv1*	Atmel *AT91RM9200*	Chipcon *CC2420*	2005
Tyndall *Tyndall Mote*	Atmel *ATmega128L*	Nordic *nRF2401*	2005
U. Karlsruhe *uPart*	Microchip *RFPIC12F675H*	Microchip *RFPIC12F675H*	2005
Yale U. *XYZ*	OKI *ML67Q5003*	Chipcon *CC2420*	2005
ETH Zurich *BTnode3*	Atmel *ATmega128L*	Zeevo *ZV4002*	2006
		Chipcon *CC1000*	
Lancaster U. *DIY*	Microchip *PIC18F252*	Radiometrix *BiM2*	2006
Texas Instruments *ez430-RF2480*	Texas Instruments *MSP430F2274*	Texas Instruments *CC2480A1*	2006
CMU *FireFly*	Atmel *ATmega1281*	Chipcon *CC2420*	2006
CSIRO *Fleck*	Atmel *ATmega128L*	Nordic *nRF903*	2006
MIT *MITes*	Nordic *nRF24E1*	Nordic *nRF24E1*	2006
Shimmer Research *Shimmer2*	Texas Instruments *MSP430F1611*	Mitsumi *WML-C46N*	2006
		Chipcon *CC2420*	
Libelium *SquidBee*	Atmel *ATmega168*	Digi *Xbee*	2006
SOWNet *T-Nodes*	Atmel *ATmega128L*	Chipcon *CC1000*	2006
Shockfish *TinyNode584*	Texas Instruments *MSP430F1611*	Semetech *XE1205*	2006
Moteiv *Tmote Invent*	Texas Instruments *MSP430F1611*	Chipcon *CC2420*	2006
U. Lucerne *WeBee*	Texas Instruments *CC2430F128*	Chipcon *CC2430F128*	2006
MechNetics *ZigBit*	Atmel *ATmega1281V*	Atmel *AT86RF230*	2006
U. Karlsruhe *zPart*	Microchip *PIC18F6720*	Chipcon *CC2420*	2006
Crossbow *Iris*	Atmel *ATmega1281*	Atmel *AT86RF230*	2007
IIT Kanpur *KMote*	Texas Instruments *MSP430F1611*	Chipcon *CC2420*	2007
ScatterWeb *MSB-430*	Texas Instruments *MSP430F1612*	Chipcon *CC1020*	2007
Intel *NeoMote*	Atmel *ATmega128L*	Chipcon *CC2420*	2007
MIT *Plug*	Atmel *AT91SAM7S64*	Chipcon *CC2500*	2007
Sensium *SPIDER*	Toumaz *TZ1030*	Toumaz *TZ1030*	2007
Atmel *AVR Raven*	Atmel *ATmega1284P*	Atmel *AT86RF230*	2008
	Atmel *ATmega3290P*		

Table A.1: Overview of the surveyed platforms and their main components *[Continued...]*

Platform	Processor	Transceiver	Release
Texas Instruments *CC2531EMK*	Texas Instruments *CC2531F256*	Texas Instruments *CC2531*	2008
UC Irvine *Eco*	Nordic *nRF24E1*	Nordic *nRF24E1*	2008
UC Berkeley *Epic*	Texas Instruments *MSP430F1611*	Chipcon *CC2420*	2008
SOWNet *G-Node G301*	Texas Instruments *MSP430F2418*	Texas Instruments *CC1101*	2008
Libelium *Waspmote*	Atmel *ATmega1281*	Digi *Xbee 802.15.4 pro*	2008
Dresden Elektronik *deUSB2400*	Atmel *AT91SAM7S256*	Atmel *AT86RF231*	2009
CEL *FreeStar PRO*	Freescale *MC13224V*	Freescale *MC13224V*	2009
Coalesences *Isense*	Jennic *JN5139*	Jennic *JN5139*	2009
Redbee *EconoTAG*	Freescale *MC13224V*	Freescale *MC13224V*	2010
Shimmer Research *Shimmer2R*	Texas Instruments *MSP430F1611*	Rowing Networks *RN-42*	2010
		Chipcon *CC2420*	
People Power *SuRF*	Texas Instruments *CC430F5137*	Texas Instruments *CC430F5137*	2010

Table A.1: Overview of the surveyed platforms and their main components.

A.2 Processing Elements

Processor	Bit-size [bit]	Max. Clock [MHz]	Flash [KB]	RAM [KB]	Min. Supply [V]
Analog Devices *ADUC841*	8	20	4	2	2.7
Atmel *AT90LS8535*	8	4	8	0.5	2.7
Atmel *AT91RM9200*	32	25	128	16	1.7
Atmel *AT91SAM7S256*	32	55	256	64	1.8
Atmel *AT91SAM7S64*	32	55	64	16	1.8
Atmel *ATmega103*	8	6	128	4	4.0
Atmel *ATmega103L*	8	4	128	4	2.7
Atmel *ATmega128*	8	16	128	4	4.5
Atmel *ATmega1281*	8	16	128	8	2.7
Atmel *ATmega1281*	8	8	128	8	1.8
Atmel *ATmega1284P*	8	20	128	16	1.8
Atmel *ATmega128L*	8	8	128	4	2.7
Atmel *AT91FR4081*	32	40	1024	136	2.7
Texas Instruments *TMS320C5416*	40	50	32	256	1.6
Atmel *ATmega163*	8	8	16	1	4.0
Atmel *ATmega168*	8	20	16	1	2.7
Texas Instruments *C8051F016*	8	25	32	2	2.7
Silicon Labs *C8051F311*	8	25	16	1	2.7
Chipcon *CC1010*	8	24	32	2	2.7
Renesas *M16C/28*	16	20	96	8	3.0

Table A.2: Overview of the processing chips used on the surveyed platforms and their main characteristics [Continued…]

Processor	Bit-size [bit]	Max. Clock [MHz]	Flash [KB]	RAM [KB]	Min. Supply [V]
Texas Instruments *CC2430F128*	8	32	128	8	2.0
Texas Instruments *CC2531F256*	8	32	256	8	2.0
Texas Instruments *CC430F5137*	16	20	32	4	1.8
Cypress *CY8C27643*	8	24	256	16	3.0
Jennic *JN5139*	32	16	192	96	2.2
Renesas *M16C/62P*	16	24	384	31	4.0
Freescale *MC13224V*	32	26	128	96	2.0
Freescale *MC68EZ328*	32	20	2048	2048	3.3
OKI *ML67Q5003*	32	60	512	32	2.3
Texas Instruments *MSP430F1232*	16	8	8	0.25	1.8
Texas Instruments *MSP430F149*	16	8	60	2	1.8
Texas Instruments *MSP430F1611*	16	8	48	10	2.0
Texas Instruments *MSP430F1612*	16	8	55	5	1.8
Texas Instruments *MSP430F169*	16	8	60	2	1.8
Texas Instruments *MSP430F2274*	16	16	32	1	1.8
Texas Instruments *MSP430F2418*	16	16	116	8	1.9
Nordic *nRF24E1*	8	20	0.5	4	1.9
Microchip *PIC16F876*	8	20	8	0.36	2.0
Microchip *PIC16F877*	8	20	8	0.36	2.0
Microchip *PIC18F252*	8	40	32	1.5	2.0
Microchip *PIC18F452*	8	40	32	1.5	2.0
Microchip *PIC18F6720*	8	25	128	3.75	2.0
Microchip *PIC18F8722*	8	42	128	3.75	2.0
Intel *PXA255*	32	400	128	3.8	1.0
Intel *PXA271*	32	520	32768	32768	1.3
Microchip *RFPIC16F675H*	8	20	1.75	0.5	2.0
Intel *SA-1100*	32	220	4096	1024	2.0
Xilinx *XC4020XLA*	0	0	0	0	3.0
Hitachi *SH-4*	32	266	32768	65536	1.8
UC Berkeley *Spec*	8	8	0	3	3.0
Zeevo *TC2001P*	32	48	512	64	1.8
Toumaz *TZ1030*	8	8	0	64	1.0
CSEM *WiseNet*	8	10	0	22	0.9
Atmel *ATmega3290P*	8	20	32	2	2.7

Table A.2: Overview of the processing chips used on the surveyed platforms and their main characteristics.

A.3 Transceivers

Transceiver	Standard	Modulation	Band [MHz]	Datarate [kbps]	TX Current [mA]
Newark *83F8851*	IrDA	PCM	IR	115.2	14
MaxStream *9xStream*		FM	915	19.2	140
Atmel *AT86RF230*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	19
Atmel *AT86RF231*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	22
Radiometrix *BiM2*		FM	433	160	20
Blueradios *BR-C11A*	Bluetooth	FHSS; GFSK	2450	721	20
Chipcon *CC1000*		FSK; OOK	433; 868; 915	76.8	27
Chipcon *CC1010*		FSK; OOK	433; 868; 915	76.8	27
Chipcon *CC1020*		FSK; OOK	433; 868; 915	153	20
Texas Instruments *CC1101*		FSK; OOK	915	500	10
Chipcon *CC2420*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	17
Rowing Networks *RN-42*	Bluetooth	FHSS; GFSK	2450	721	50
Mitsumi *WML-C46N*	Bluetooth	GFSK; Pi/4-DQPSK; 8-DPSK	2450	721	50
Chipcon *CC2430F128*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	27
Texas Instruments *CC2480A1*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	27
Texas Instruments *CC2531*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	34
Texas Instruments *CC430F5137*		FSK; GFSK; MSK; OOK	315; 433; 868; 915	500	35
Ember *EM250*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	33
Telegesis *ETRX2*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	42
Jennic *JN5139*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	34
National Semiconductor *LMX3162*		FSK	2450	1000	40
National Semiconductor *LMX9820*	Bluetooth	FHSS; GFSK	2450	721	56
Freescale *MC13224V*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	31
Nordic *nRF2401*		GFSK	2450	1000	13
Nordic *nRF24E1*		GFSK	2450	1000	13
Nordic *nRF903*		GFSK	433; 868; 915	76.8	30
Proxim *RangeLAN2*	IEEE 802.11	DSSS; DBPSK; DQPSK	2450	1600	300
Conexant *RDSSS9M*	DECT	FHSS; GFSK	915	128	13
Microchip *RFPIC12F675H*		ASK; FSK	868; 915	40	20
Ericsson *ROK101007*	Bluetooth	FHSS; GFSK	2450	721	26
Ericsson *ROK101008*	Bluetooth	FHSS; GFSK	2450	721	26
UC Berkeley *Spec*		FSK	915	19.2	1
Zeevo *TC2001P*	Bluetooth	FHSS; GFSK	2450	721	20
Infineon *TDA5250*		ASK; FSK	868	50	12

Table A.3: Overview of the transceiver chips used on the surveyed platforms and their main characteristics [Continued...]

Transceiver	Standard	Modulation	Band [MHz]	Datarate [kbps]	TX Current [mA]
RFM *TR1000*		ASK; OOK	915	115.2	12
RFM *TR1001*		ASK; OOK	868	115.2	12
RFM *TR3001*		ASK; OOK	315	115.2	10
Radiometrix *TX1H*		FM	173	10	80
Toumaz *TZ1030*		FSK	868; 915	50	3
Sensoria *WINS NG RF 2.0*		FSK	2450	56	30
CSEM *WiseNET*		FSK; OOK	433; 868	100	24
Mitsumi *WML-C10AHR*	Bluetooth	FHSS; GFSK	2450	721	60
Digi *Xbee*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	45
Digi *Xbee 802.15.4 pro*	IEEE 802.15.4	DSSS; O-QPSK	2450	250	45
Semetech *XE1205*		FSK	433; 868; 915	305	62
Zeevo *ZV4002*	Bluetooth	FHSS; GFSK	2450	721	22

Table A.3: Overview of the transceiver chips used on the surveyed platforms and their main characteristics.

Publications

Parts of this dissertation have already been published in the following publications. All collaborators of the work covered by this dissertation are also co-authors of the below listed joint publications.

Conference and Workshop Proceedings

- J.-H. Hauer, V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "A Component Framework for Content-based Publish/Subscribe in Sensor Networks", In Proc. of 5th European Conference on Wireless Sensor Networks (EWSN), Bologna, Italy, January 2008 Springer.

- K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis, "Integrating Concurrency Control and Energy Management in Device Drivers", In Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP 2007), Stevenson, WA, USA, October 2007.

- V. Handziski, A. Köpke, A. Willig, and A. Wolisz, "TWIST: A Scalable and Reconfigurable Testbed for Wireless Indoor Experiments with Sensor Network", In Proc. of the 2nd Intl. Workshop on Multi-hop Ad Hoc Networks: from Theory to Reality, (RealMAN 2006), Florence, Italy, May 2006.

- V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler, "Flexible Hardware Abstraction for Wireless Sensor Networks", In Proc. of 2nd European Workshop on Wireless Sensor Networks (EWSN 2005), Istanbul, Turkey, February 2005.

- V. Handziski, J. Polastre, J. -H. Hauer, and C. Sharp, "Flexible hardware abstraction of the TI MSP430 microcontroller in TinyOS", In Proc. of SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems, pp. 277-278, Baltimore, MD, USA, November 2004 ACM Press.

- V. Handziski, A. Köpke, H. Karl, C. Frank, and W. Drytkiewicz, "Improving the Energy Efficiency of Directed Diffusion Using Passive Clustering", In H. Karl, A. Willig, and A. Wolisz, editors, Proc. of 1st European Workshop on Wireless Sensor Networks (EWSN), Volume 2920 of LNCS, Berlin, Germany, January 2004 Springer.

Technical Reports

- V. Handziski, C. Donzelli, and I. Antonova, "RESTful Platform for Federating WSN Testbeds", Technical Report TKN-10-001, Telecommunication Networks Group, Technical University Berlin, January 2010.
- P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz, "T2: A Second Generation OS For Embedded Sensor Networks", Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, November 2005.
- V. Handziski, C. Frank, and H.Karl, "Service Discovery in Wireless Sensor Networks", Technical Report TKN-04-006, Telecommunication Networks Group, Technische Universität Berlin, March 2004.
- V. Handziski, A. Köpke, C. Frank, and H.Karl, "Semantic Addressing for Wireless Sensor Networks", Technical Report TKN-04-005, Telecommunication Networks Group, Technische Universität Berlin, May 2004.
- A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl, "Structuring the Information Flow in Component-Based Protocol Implementations for Wireless Sensor Nodes", In Proc. of Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN), Technical Report TKN-04-001, Telecommunication Networks Group, Technische Universität Berlin, pp. 41-45, Berlin, Germany, January 2004.
- V. Handziski, H. Karl, A. Köpke, and A. Wolisz, "A common wireless sensor network architecture?", In H. Karl, editor, Proc. 1. GI/ITG Fachgespräch "Sensornetze" (Technical Report TKN-03-012 of the Telecommunications Networks Group, Technische Universität Berlin), pp. 10-17, Berlin, Germany, July 2003.

Bibliography

[1] Domain-specific software architecture (DSSA) frequently asked questions (FAQ). *SIGSOFT Softw. Eng. Notes*, 19:52–56, April 1994. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/181628.181639.

[2] 6LoWPAN. Ipv6 over low power wpan. URL http://www.ietf.org/dyn/wg/charter/6lowpan-charter.html.

[3] R. Adler, M. Flanigan, J. Huang, R. Kling, N. Kushalnagar, L. Nachman, C.-Y. Wan, and M. Yarvis. Intel mote 2: An advanced platform for demanding sensor network applications. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 298–298, New York, NY, USA, 2005. ACM. ISBN 1-59593-054-X. doi: http://doi.acm.org/10.1145/1098918.1098963.

[4] I. Akyildiz, W. Su, Y. Sankarasubramaniam, and E. Cayirci. A survey on sensor networks. *Communications Magazine, IEEE*, 40(8):102–114, Aug 2002. ISSN 0163-6804. doi: 10.1109/MCOM.2002.1024422.

[5] A. Albrecht and J. Gaffney, J.E. Software function, source lines of code, and development effort prediction: A software science validation. *Software Engineering, IEEE Transactions on*, SE-9(6):639 – 648, nov. 1983. ISSN 0098-5589. doi: 10.1109/TSE.1983.235271.

[6] M. Banâtre, P. J. Marrón, A. Ollero, and A. Wolisz, editors. *Cooperating Embedded Systems and Wireless Sensor Networks*. John Wiley & Sons, Inc., 2008.

[7] L. Bass. *Software Architecture in Practice*. Addison-Wesley, Boston, 2003. ISBN 0321154959.

[8] D. Batory, L. Coglianese, M. Goodwin, and S. Shafer. Creating reference architectures: an example from avionics. *SIGSOFT Softw. Eng. Notes*, 20(SI):27–37, 1995. ISSN 0163-5948. doi: http://doi.acm.org/10.1145/223427.211786.

[9] N. Beck and I. Johnson. Shaping tinyos to deal with evolving device architectures: experiences porting tinyos-2.0 to the chipcon cc2430. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 83–87, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-694-3. doi: http://doi.acm.org/10.1145/1278972.1278994.

[10] C. Becker, G. Schiele, H. Gubbels, and K. Rothermel. Base - a micro-broker-based middleware for pervasive computing. In *Pervasive Computing and Communications, 2003. (PerCom 2003). Proceedings of the First IEEE International Conference on*, pages 443 – 451, 2003. doi: 10.1109/PERCOM.2003.1192769.

[11] L. Benini, A. Bogliolo, and G. De Micheli. A survey of design techniques for system-level dynamic power management. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 8(3):299–316, June 2000. ISSN 1063-8210. doi: 10.1109/92.845896.

[12] M. Berekovic. State of the art in wireless sensor nodes, Mar. 2007. URL www.igd.fhg.de/igd-a1/wasp/public/WASP_Technology_SoA_WSN.pdf.

[13] T. Berners-Lee, R. Fielding, and L. Masinter. Uniform resource identifiers (URI): Generic syntax, 2005.

[14] J. Beutel. The sensor network museum. URL http://www.snm.ethz.ch/Main/HomePage.

[15] 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. *Mob. Netw. Appl.*, 10(4):563–579, 2005. ISSN 1383-469X. doi: http://doi.acm.org/10.1145/1160162.1160178.

[16] J. Blumenthal, M. Handy, F. Golatowski, M. Haase, and D. Timmermann. Wireless sensor networks - new challenges in software engineering. In *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, volume 1, pages 551–556 vol.1, Sept. 2003. doi: 10.1109/ETFA.2003.1247755.

[17] T. Bokareva. Mini hardware survey. URL http://www.cse.unsw.edu.au/~sensar/hardware/hardware_survey.html.

[18] C. Borcea, C. Intanagonwiwat, A. Saxena, and L. Iftode. Self-routing in pervasive computing environments using smart messages. In *Proceedings of the First IEEE International Conference on Pervasive Computing and Communications*, page 87. IEEE Computer Society, 2003. ISBN 0-7695-1893-1.

[19] A. Boulis and M. B. Srivastava. A framework for efficient and programmable sensor networks. In *Proceedings of the 5th International Conference on Open Architectures and Network Programming (OPENARCH 2002)*, pages 117–128, June 2002.

[20] R. Braden. Requirements for Internet Hosts - Communication Layers. RFC 1122 (Standard), Oct. 1989. URL http://www.ietf.org/rfc/rfc1122.txt.

[21] N. Brouwers, P. Corke, and K. Langendoen. Darjeeling, a java compatible virtual machine for microcontrollers. In *Companion '08: Proceedings of the ACM/IFIP/USENIX Middleware '08 Conference Companion*, pages 18–23, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-369-3. doi: http://doi.acm.org/10.1145/1462735.1462740.

[22] Cacti. URL http://www.cacti.net.

[23] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The liteos operating system: Towards unix-like abstractions for wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 233–244,

Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3157-1. doi: http://dx.doi.org/10.1109/IPSN.2008.54.

[24] A. Carzaniga. *Architectures for an Event Notification Service Scalable to Wide-area Networks*. PhD thesis, Politecnico di Milano, Milano, Italy, Dec. 1998. URL http://www.cs.colorado.edu/~carzanig/papers/.

[25] A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *NSF Workshop on an Infrastructure for Mobile and Wireless Systems*, number 2538 in Lecture Notes in Computer Science, Scottsdale, Arizona, Oct. 2001. Springer-Verlag.

[26] A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems (TOCS)*, 19(3), 2001. ISSN 0734-2071. doi: http://doi.acm.org/10.1145/380749.380767.

[27] K. K. Chang and D. Gay. Language support for interoperable messaging in sensor networks. In *SCOPES '05: Proceedings of the 2005 workshop on Software and compilers for embedded systems*, pages 1–9, New York, NY, USA, 2005. ACM. ISBN 1-59593-207-0. doi: http://doi.acm.org/10.1145/1140389.1140390.

[28] C.-K. Chau, M. H. Wahab, F. Qin, Y. Wang, and Y. Yang. Battery recovery aware sensor networks. In *WiOPT'09: Proceedings of the 7th international conference on Modeling and Optimization in Mobile, Ad Hoc, and Wireless Networks*, pages 203–211, Piscataway, NJ, USA, 2009. IEEE Press. ISBN 978-1-4244-4919-4.

[29] C.-Y. Chen, Y.-T. Chen, Y.-H. Tu, S.-Y. Yang, and P. Chou. EcoSpire: An application development kit for an ultra-compact wireless sensing system. *Embedded Systems Letters, IEEE*, 1(3):65–68, 2009. ISSN 1943-0663. doi: 10.1109/LES.2009.2037984.

[30] Chipcon Inc. CC2420 data sheet, 2003. URL http://www.chipcon.com/files/CC2420_Data_Sheet_1_0.pdf.

[31] D. Chu, K. Lin, A. Linares, G. Nguyen, and J. M. Hellerstein. Sdlib: a sensor network data and communications library for rapid and robust application development. In *IPSN '06: Proceedings of the fifth international conference on Information processing in sensor networks*, New York, NY, USA, 2006. ACM Press. ISBN 1-59593-334-4. doi: http://doi.acm.org/10.1145/1127777.1127843.

[32] B. N. Chun, P. Buonadonna, A. AuYoung, C. Ng, D. C. Parkes, J. Shneidman, A. C. Snoeren, and A. Vahdat. Mirage: A microeconomic resource allocation system for sensornet testbeds. In *Proceedings of the 2nd IEEE Workshop on Embedded Networked Sensors*, May 2005.

[33] E. Clarke, O. Grumberg, and D. Peled. *Model checking*. MIT Press, 1999. ISBN 9780262032704.

[34] W. Cleveland and S. Devlin. Locally weighted regression: an approach to regression analysis by local fitting. *Journal of the American Statistical Association*, 83(403):596–610, 1988. ISSN 0162-1459.

[35] D. Crockford. The application/json media type for JavaScript object notation (JSON), 2006.

[36] Crossbow. Stargate. URL http://www.xbow.com/Products/XScale.htm.

[37] D. Culler, P. Dutta, C. T. Ee, R. Fonseca, J. Hui, P. Levis, J. Polastre, S. Shenker, I. Stoica, G. Tolle, and J. Zhao. Towards a sensor network architecture: lowering the waistline. In *HOTOS'05: Proceedings of the 10th conference on Hot Topics in Operating Systems*, pages 24–24, Berkeley, CA, USA, 2005. USENIX Association.

[38] L. Daigle, D. Van Gulik, R. Iannella, and P. Faltstrom. Uniform resource names (URN) namespace definition mechanisms, 2002.

[39] C. Decker. Wireless sensor network platforms. URL http://www.docstoc.com/docs/32679903/Wireless-Sensor-Network-(WSN)-Platforms-A-Brief-History-Early.

[40] T. Dishongh and M. McGrath. *Wireless Sensor Networks for Healthcare Applications*. Artech House, 2009. ISBN 9781596933057.

[41] W. Dong, Y. Liu, X. Wu, L. Gu, and C. Chen. Elon: enabling efficient and long-term reprogramming for wireless sensor networks. In *SIGMETRICS '10: Proceedings of the ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 49–60, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0038-4. doi: http://doi.acm.org/10.1145/1811039.1811046.

[42] A. Dunkels, B. Grönvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proceedings of the First IEEE Workshop on Embedded Networked Sensors 2004 (IEEE EmNetS-I)*, Nov. 2004.

[43] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: Simplifying event-driven programming of memory-constrained embedded systems. In *Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems (SenSys 2006)*, Nov. 2006.

[44] A. Dunkels, F. Österlind, and Z. He. An adaptive communication architecture for wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 335–349, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-763-6. doi: http://doi.acm.org/10.1145/1322263.1322295. URL http://doi.acm.org/10.1145/1322263.1322295.

[45] C. T. Ee, R. Fonseca, S. Kim, D. Moon, A. Tavakoli, D. Culler, S. Shenker, and I. Stoica. A modular network layer for sensornets. In *Proceedings of the 7th Symposium on Operating Systems Design and Implementation (OSDI 2006)*, 2006.

[46] D. R. Engler, M. F. Kaashoek, and J. O'Toole, Jr. Exokernel: an operating system architecture for application-level resource management. In *Proceedings of the fifteenth ACM symposium on Operating systems principles*, pages 251–266. ACM Press, 1995. ISBN 0-89791-715-4. doi: http://doi.acm.org/10.1145/224056.224076.

[47] E. R. C. (ERC). The european table of frequency allocations and utilisations, covering the frequency range 9 khz to 275 gh. In *European Conference of Postal and Telecommunications Administrations (CEPT)*, Lisboa, Jan. 2002.

[48] S. Escolar, J. Carretero, F. Isaila, and F. Garcia. A driver model based on linux for tinyos. pages 361–364, July 2007. doi: 10.1109/SIES.2007.4297362.

[49] *BTNodes rev. 3*. ETH Zürich, 2005.

[50] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR)*, 35(2):114–131, 2003. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/857076.857078.

[51] EYES Project. Energy efficient wireless sensor networks (EYES) project, ist-2001-34734. URL http://www.eyes.eu.org.

[52] R. Falk, H.-J. Hof, U. Meyer, C. Niedermeier, R. Sollacher, and N. Vicari. From academia to the field: Wireless sensor networks for industrial use. In *Proceedings of 7. GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze"*, Berlin, Germany, Sept. 2008. Freie Universität Berlin, Institute of Computer Science. URL ftp://ftp.inf.fu-berlin.de/pub/reports/tr-b-08-12.pdf. Technical Report B 08-12.

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

[54] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, 1999.

[55] R. S. S. Filho and D. F. Redmiles. Striving for versatility in publish/subscribe infrastructures. In *SEM '05: Proceedings of the 5th international workshop on Software engineering and middleware*, New York, NY, USA, 2005. ISBN 1-59593-204-4. doi: http://doi.acm.org/10.1145/1108473.1108478.

[56] FIRE Project. Future internet research & experimentation (FIRE) project. URL http://www.ict-fire.eu/home.html.

[57] K. Fowler. The future of sensors and sensor networks survey results projecting the next 5 years. In *Sensors Applications Symposium, 2009. SAS 2009. IEEE*, pages 1 –6, feb. 2009. doi: 10.1109/SAS.2009.4801766.

[58] Freaklabs. Zigbee/802.15.4 chip comparison guide. URL http://freaklabs.org/index.php/Articles/Zigbee/Zigbee-Chip-Comparison.html.

[59] Free Software Foundation. GNU binary utilities. URL http://www.gnu.org/software/binutils/.

[60] N. Freed and B. N. Multipurpose internet mail extensions (MIME) part two: Media types, 1996.

[61] L. F. Friedrich, J. Stankovic, M. Humphrey, M. Marley, and J. Haskins. A survey of configurable, component-based operating systems for embedded applications. *IEEE Micro*, 21(3):54–68, 2001. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/40.928765.

[62] W. F. Fung, D. Sun, and J. Gehrke. Cougar: the network is the database. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 621–621. ACM Press, 2002. ISBN 1-58113-497-5. doi: http://doi.acm.org/10.1145/564691.564775.

[63] J. Gao, H. Tsao, and Y. Wu. *Testing and quality assurance for component-based software*. Artech House Computing Library. Artech House, 2003. ISBN 9781580534802. URL http://books.google.com/books?id=VoCX09hOsCoC.

[64] D. Garlan, S. Khersonsky, and J. S. Kim. Model checking publish-subscribe systems. In *Proceedings of the 10th international conference on Model checking software*, SPIN'03, pages 166–180, Berlin, Heidelberg, 2003. Springer-Verlag. ISBN 3-540-40117-2. URL http://portal.acm.org/citation.cfm?id=1767111.1767122.

[65] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, pages 1–11, New York, NY, USA, 2003. ACM Press. ISBN 1-58113-662-5. doi: http://doi.acm.org/10.1145/781131.781133.

[66] D. Gay, P. Levis, D. Culler, and E. Brewer. nesC 1.2 language reference manual, 2006. URL http://nescc.cvs.sourceforge.net/*checkout*/nescc/nesc/doc/ref.pdf.

[67] Z. Ge, P. Ji, J. Kurose, and D. Towsley. Matchmaker: Signaling for dynamic publish/subscribe applications. In *ICNP '03: Proceedings of the 11th IEEE International Conference on Network Protocols*, Washington, DC, USA, 2003. ISBN 0-7695-2024-3.

[68] GENI Project. URL http://www.geni.net/.

[69] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of USENIX General Track 2004*, Boston, MA, USA, June 2004.

[70] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 201–213, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-879-2. doi: http://doi.acm.org/10.1145/1031495.1031519.

[71] O. Gnawali, R. Fonseca, K. Jamieson, D. Moss, and P. Levis. Collection tree protocol. In *SenSys '09: Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, pages 1–14, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2. doi: http://doi.acm.org/10.1145/1644038.1644040.

[72] Gnutella. URL http://gnutella.wego.com.

[73] Google Inc. PubSubHubbub. URL http://code.google.com/p/pubsubhubbub.

[74] R. Govindan, J. M. Hellerstein, W. Hong, S. Madden, M. Franklin, and S. Shenker. The sensor network as a database. Technical Report 02-771, USC/Information Sciences Institute, Sept. 2002.

[75] P. Grace, G. Coulson, G. Blair, B. Porter, and D. Hughes. Dynamic reconfiguration in sensor middleware. In *Proceedings of the international workshop on Middleware for sensor networks*, MidSens '06, pages 1–6, New York, NY, USA, 2006. ACM. ISBN 1-59593-424-3. doi: http://doi.acm.org/10.1145/1176866.1176867.

[76] S. Greenberg and C. Fitchett. Phidgets: easy development of physical interfaces through physical widgets. In *UIST '01: Proceedings of the 14th annual ACM symposium on User interface software and technology*, pages 209–218, New York, NY, USA, 2001. ACM. ISBN 1-58113-438-X. doi: http://doi.acm.org/10.1145/502348.502388.

[77] L. Gu and J. A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 1–14, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. doi: http://doi.acm.org/10.1145/1182807.1182809.

[78] T. R. Halfhi. Philips challenges 8-bit MCUs. *Microprocessor Report*, 2005. URL http://ics.nxp.com/support/documents/microcontrollers/pdf/article.challenge.8-bit.mcu.pdf.

[79] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, D. Culler, and D. Gay. TEP2: hardware abstraction architecture. URL http://www.tinyos.net/tinyos-2.1.0/doc/html/tep2.html.

[80] V. Handziski, H. Karl, A. Köpke, and A. Wolisz. A common wireless sensor network architecture? In H. Karl, editor, *Proceedings 1. GI/ITG Fachgespräch "Sensornetze" (Technical Report TKN-03-012 of the Telecommunications Networks Group, Technische Universität Berlin*, pages 10–17, Berlin, Germany, July 2003.

[81] V. Handziski, J. Polastre, J.-H. Hauer, and C. Sharp. Flexible hardware abstraction of the TI MSP430 microcontroller in TinyOS. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, New York, NY, USA, 2004. ACM Press. ISBN 1-58113-879-2. doi: http://doi.acm.org/10.1145/1031495.1031534.

[82] V. Handziski, J. Polastre, J.-H. Hauer, C. Sharp, A. Wolisz, and D. Culler. Flexible hardware abstraction for wireless sensor networks. In *Proceedings of Second European Workshop on Wireless Sensor Networks (EWSN 2005)*, Istanbul, Turkey, Feb. 2005.

[83] J.-J. Hauer. Service discovery in wireless sensor networks using publish/subscribe middleware. Master's thesis, Telecommunication Networks Group, Technische Universität Berlin, 2005.

[84] P. Havinga, L. Evers, J. Wu, H. Karl, A. Kopke, V. Handziski, and M. Zorzi. Snapshots of the EYES project. pages 45–52, May 2004. doi: 10.1109/IWWAN.2004.1525539.

[85] J. Heidemann, F. Silva, and D. Estrin. Matching data dissemination algorithms to application requirements. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, New York, NY, USA, 2003. ISBN 1-58113-707-9. doi: http://doi.acm.org/10.1145/958491.958517.

[86] H. Heinecke, K. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J. Maté, K. Nishikawa, and T. Scharnhorst. *Automotive open system architecture-an industry-wide initiative to manage the complexity of emerging automotive E/E architectures*. Society of Automotive Engineers, 400 Commonwealth Dr, Warrendale, PA, 15096, USA,, 2004.

[87] W. B. Heinzelman, A. L. Murphy, H. S. Carvalho, and M. A. Perillo. Middleware to support sensor network applications. *IEEE Network*, 18(1), 2004.

[88] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proceedings of the ninth international conference on Architectural support for programming languages and operating systems (ASPL 2000)*, 2000. ISBN 1-58113-317-0. doi: http://doi.acm.org/10.1145/378993.379006.

[89] J. Hill, M. Horton, R. Kling, and L. Krishnamurthy. The platforms enabling wireless sensor networks. *Commun. ACM*, 47(6):41–46, 2004. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/990680.990705.

[90] J. L. Hill. *System architecture for wireless sensor networks*. PhD thesis, 2003. Adviser-Culler, David E.

[91] J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002. ISSN 0272-1732. doi: http://dx.doi.org/10.1109/MM.2002.1134340.

[92] T. W. Hnat, T. I. Sookoor, P. Hooimeijer, W. Weimer, and K. Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 225–238, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. doi: http://doi.acm.org/10.1145/1460412.1460435.

[93] Y. hua Chu, S. G. Rao, and H. Zhang. A case for end system multicast (keynote address). In *Proceedings of the 2000 ACM SIGMETRICS international conference on Measurement and modeling of computer systems*, pages 1–12. ACM Press, 2000. ISBN 1-58113-194-1. doi: http://doi.acm.org/10.1145/339331.339337.

[94] IANA. URL http://www.iana.org/assignments/urn-namespaces/.

[95] IEEE. IEEE standard for a smart transducer interface for sensors and actuators - transducer to microprocessor communication protocols and transducer electronic data sheet (TEDS) formats. *IEEE Std 1451.2-1997*, 1998. doi: 10.1109/IEEESTD.1998.88285.

[96] IEEE. IEEE standard for information technology- portable operating system interface (POSIX) base specifications, issue 7. *IEEE Std 1003.1-2008 (Revision of IEEE Std 1003.1-2004)*, pages c1–3826, 1 2008. doi: 10.1109/IEEESTD.2008.4694976.

[97] IETF ROLL Working Group. Routing over low power and lossy networks. URL http://www.ietf.org/dyn/wg/charter/roll-charter.html.

[98] T. Imielinski and S. Goel. Dataspace-querying and monitoring deeply networked collections in physical space. In *Proceedings of the ACM international workshop on Data engineering for wireless and mobile access*, pages 44–51. ACM Press, 1999. ISBN 1-58113-175-5. doi: http://doi.acm.org/10.1145/313300.313341.

[99] Institute of Systems for Complex Automation. Overview of the elements database, 2008. URL http://isca.su/index.php?option=com_content&task=view&id=24&Itemid=29.

[100] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking (TON)*, 11(1):2–16, 2003. ISSN 1063-6692. doi: http://doi.acm.org/10.1145/638334.638335.

[101] ISA100. ISA100 wireless compliance institute. URL http://www.isa.org/Content/NavigationMenu/Technical_Information/ASCI/ISA100_Wireless_Compliance_Institute/ISA100_Wireless_Compliance_Institute.htm.

[102] ISO. Basic reference model: The basic model. ISO/IEC 7498 Part I, 1994. URL http://standards.iso.org/ittf/PubliclyAvailableStandards/s020269_ISO_IEC_7498-1_1994(E).zip.

[103] ISO. Systems and software engineering—recommended practice for architectural description of software-intensive systems. *ISO/IEC 42010 IEEE Std 1471-2000 First edition 2007-07-15*, pages c1–24, 15 2007. doi: 10.1109/IEEESTD.2007.386501.

[104] X. Jiang, J. Polastre, and D. Culler. Perpetual environmentally powered sensor networks. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 463–468, april 2005. doi: 10.1109/IPSN.2005.1440974.

[105] X. Jiang, J. Taneja, J. Ortiz, A. Tavakoli, P. Dutta, J. Jeong, D. Culler, P. Levis, and S. Shenker. An architecture for energy management in wireless sensor networks. *SIGBED Rev.*, 4(3):31–36, 2007. doi: http://doi.acm.org/10.1145/1317103.1317109.

[106] X. Jiang, S. Dawson-Haggerty, P. Dutta, and D. Culler. Design and implementation of a high-fidelity AC metering network. In *IPSN '09: Proceedings of the 2009 International Conference on Information Processing in Sensor Networks*, pages 253–264, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-1-4244-5108-1.

[107] R. E. Johnson. Frameworks = (components + patterns). *Commun. ACM*, 40:39–42, October 1997. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/262793.262799.

[108] J. A. Jun. Wireless sensor network platforms. URL http://mnet.skku.ac.kr/data/2006data/KRnet2006/A/A2-2.pdf.

[109] O. Kasten and J. Beutel. BTnode rev2.2. URL http://www.inf.ethz.ch/vs/res/proj/smart-its/btnode.html.

[110] Kenai. Sun Cloud API. URL http://kenai.com/projects/suncloudapis/.

[111] L. Kleinrock. Nomadic computing—an opportunity. *SIGCOMM Comput. Commun. Rev.*, 25(1):36–40, 1995. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/205447.205450.

[112] K. Klues, G. Hackmann, O. Chipara, and C. Lu. A component-based architecture for power-efficient media access control in wireless sensor networks. In *Proceedings of the 5th international conference on Embedded networked sensor systems*, SenSys '07, pages 59–72, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-763-6. doi: http://doi.acm.org/10.1145/1322263.1322270. URL http://doi.acm.org/10.1145/1322263.1322270.

[113] K. Klues, V. Handziski, C. Lu, A. Wolisz, D. Culler, D. Gay, and P. Levis. Integrating concurrency control and energy management in device drivers. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 251–264, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-591-5. doi: http://doi.acm.org/10.1145/1294261.1294286.

[114] M. Kohvakka, M. Hannikainen, and T. Hamalainen. Wireless sensor prototype platform. In *Industrial Electronics Society, 2003. IECON '03. The 29th Annual Conference of the IEEE*, volume 2, pages 1499 – 1504 Vol.2, nov. 2003. doi: 10.1109/IECON.2003.1280279.

[115] A. Köpke, V. Handziski, J.-H. Hauer, and H. Karl. Structuring the information flow in component-based protocol implementations for wireless sensor nodes. In *Proceedings Work-in-Progress Session of the 1st European Workshop on Wireless Sensor Networks (EWSN)*, Technical Report TKN-04-001 of Technical University Berlin, Telecommunication Networks Group, pages 41–45, Berlin, Germany, Jan. 2004.

[116] J. Koshy and R. Pandey. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM. ISBN 1-59593-054-X. doi: http://doi.acm.org/10.1145/1098918.1098945.

[117] C. W. Krueger. Software reuse. *ACM Computer Survey*, 24(2):131–183, 1992. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/130844.130856.

[118] G. S. Kubota. Matching device drivers with embedded hardware. *RTC Magazine*, April 2005. URL http://rtcmagazine.com/articles/view/100329.

[119] J. Kulik, W. Heinzelman, and H. Balakrishnan. Negotiation-based protocols for disseminating information in wireless sensor networks. *Wireless Networks*, 8(2/3), 2002. ISSN 1022–0038. doi: http://dx.doi.org/10.1023/A:1013715909417.

[120] M. Kwon and S. Fahmy. Topology-aware overlay networks for group communication. In *Proceedings of the 12th international workshop on Network and operating systems support for digital audio and video*, pages 127–136. ACM Press, 2002. ISBN 1-58113-512-2. doi: http://doi.acm.org/10.1145/507670.507688.

[121] A. Lachenmann, P. Marron, D. Minder, M. Gauger, O. Saukh, and K. Rothermel. TinyXXL: Language and runtime support for cross-layer interactions. In *Sensor and Ad Hoc Communications and Networks, 2006. SECON '06. 2006 3rd Annual IEEE Communications Society on*, volume 1, pages 178–187, 2006. doi: 10.1109/SAHCN.2006.288422.

[122] A. Lachenmann, P. J. Marrón, D. Minder, and K. Rothermel. Meeting lifetime goals with energy levels. In *SenSys '07: Proceedings of the 5th international conference on Embedded networked sensor systems*, pages 131–144, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-763-6. doi: http://doi.acm.org/10.1145/1322263.1322277.

[123] M. Lauer. Building linux distributions with bitbake and openembedded. In *Proceedings of the Free and Open Source Developers' European Meeting (FOSDEM 2005)*, Brussels, Belgium, Feb. 2005.

[124] E. Lee. What's ahead for embedded software? *Computer*, 33(9):18 –26, Sept. 2000. ISSN 0018-9162. doi: 10.1109/2.868693.

[125] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM. ISBN 1-58113-574-2. doi: http://doi.acm.org/10.1145/605397.605407.

[126] P. Levis, N. Patel, D. Culler, and S. Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[127] P. Levis, D. Gay, V. Handziski, J.-H.Hauer, B.Greenstein, M.Turon, J.Hui, K.Klues, C.Sharp, R.Szewczyk, J.Polastre, P.Buonadonna, L.Nachman, G.Tolle, D.Culler, and A.Wolisz. T2: A second generation OS for embedded sensor networks. Technical Report TKN-05-007, Telecommunication Networks Group, Technische Universität Berlin, Nov. 2005.

[128] P. Levis, E. Brewer, D. Culler, D. Gay, S. Madden, N. Patel, J. Polastre, S. Shenker, R. Szewczyk, and A. Woo. The emergence of a networking primitive in wireless sensor networks. *Commun. ACM*, 51(7):99–106, 2008. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/1364782.1364804.

[129] J. Lifton, M. Broxton, and J. A. Paradiso. Experiences and directions in pushpin computing. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 57, Piscataway, NJ, USA, 2005. IEEE Press. ISBN 0-7803-9202-7.

[130] K. Lin and P. Levis. Data discovery and dissemination with DIP. In *Proceedings of the 7th international conference on Information processing in sensor networks*, IPSN '08, pages 433–444, Washington, DC, USA, 2008. IEEE Computer Society. ISBN 978-0-7695-3157-1. doi: http://dx.doi.org/10.1109/IPSN.2008.17.

[131] K. Lind and R. Heldal. Categorization of real-time software components for code size estimation. In *ESEM '10: Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, pages 1–10, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0039-1. doi: http://doi.acm.org/10.1145/1852786.1852821.

[132] T. Lindholm and F. Yellin. *Java Virtual Machine Specification*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999. ISBN 0201432943.

[133] B. Lo. Hardware platforms. URL http://vip.doc.ic.ac.uk/bsn/m206.html.

[134] C. Lynch and F. O. Reilly. Processor choice for wireless sensor networks. In *REAL-WSN'05: Workshop on Real-World Wireless Sensor Networks*, 2005.

[135] J. Lynch and K. Loh. A Summary Review of Wireless Sensors and Sensor Networks for Structural Health Monitoring. *The Shock and Vibration Digest*, 38(2):91–128, 2006.

[136] S. Madani, S. Mahlknecht, and J. Glaser. A step towards standardization of wireless sensor networks: A layered protocol architecture perspective. In *Sensor Technologies and Applications, 2007. SensorComm 2007. International Conference on*, pages 82–87, Oct. 2007. doi: 10.1109/SENSORCOMM.2007.4394902.

[137] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. *ACM SIGOPS Operating Systems Review*, 36(SI): 131–146, 2002. ISSN 0163-5980. doi: http://doi.acm.org/10.1145/844128.844142.

[138] A. Massa. *Embedded Software Development with eCos*. Prentice Hall Professional Technical Reference, 2002. ISBN 0130354732.

[139] T. May, S. Dunning, and J. Hallstrom. An RPC design for wireless sensor networks. Nov. 2005. doi: 10.1109/MAHSS.2005.1542785.

[140] W. P. McCartney and N. Sridhar. Abstractions for safe concurrent programming in networked embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 167–180, New York, NY, USA, 2006. ACM. ISBN 1-59593-343-3. doi: http://doi.acm.org/10.1145/1182807.1182825.

[141] M. McIlroy, J. Buxton, P. Naur, and B. Randell. Mass-Produced Software Components. *Software Engineering Concepts and Techniques (1968*, pages 88–98, 1968.

[142] Micrel Lab, University of Bologna. Hardware survey. URL http://www-micrel.deis.unibo.it/sitonew/research/Ami/node_survey/hardware_survey.htm.

[143] P. Millard, P. Saint-Andre, and R. Meijer. XEP-0060: Publish-subscribe. *Jabber Software Foundation*, 2006. URL http://xmpp.org/extensions/xep-0060.html.

[144] D. Mills. Internet time synchronization: the network time protocol. *Communications, IEEE Transactions on*, 39(10):1482–1493, Oct. 1991. ISSN 0090-6778. doi: 10.1109/26.103043.

[145] P. Mochel. The linux kernel device model. Proceedings of the Linux.Conf.Au conference (LCA 2003), 2003. URL http://conf.linux.org.au.

[146] J. Mooney. Strategies for supporting application portability. *IEEE Computer Magazine*, 23(11):59–70, Nov. 1990. ISSN 0018-9162. doi: 10.1109/2.60881.

[147] Moteiv Corporation. Tmote invent user guide. URL http://sentilla.com/files/pdf/eol/tmote-invent-user-guide.pdf.

[148] G. Mühl, L. Fiege, and A. P. Buchmann. Filter similarities in content-based publish/subscribe systems. In H. Schmeck, T. Ungerer, and L. Wolf, editors, *International Conference on Architecture of Computing Systems (ARCS)*, volume 2299 of *Lecture Notes in Computer Science*, pages 224–238, Karlsruhe, Germany, 2002. Springer-Verlag. URL http://link.springer.de/link/service/series/0558/bibs/2299/22990224.htm.

[149] G. Mühl, L. Fiege, F. Gartner, and A. Buchmann. Evaluating advanced routing algorithms for content-based publish/subscribe systems. In *Proceedings. 10th IEEE International Symposium onModeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2002)*, pages 167–176. IEEE, 2002.

[150] J. Murray. *Inside Microsoft Windows CE*. Microsoft Press, 1998. ISBN 1572318546.

[151] L. Nachman, R. Kling, R. Adler, J. Huang, and V. Hummel. The intel mote platform: a bluetooth-based sensor network for industrial monitoring. pages 437–442, Apr. 2005. doi: 10.1109/IPSN.2005.1440968.

[152] L. Nachman, J. Huang, J. Shahabdeen, R. Adler, and R. Kling. IMOTE2: Serious computation at the edge. pages 1118–1123, Aug. 2008. doi: 10.1109/IWCMC.2008.194.

[153] Napster, Inc. URL http://www.napster.com.

[154] NetBSD. The NetBSD project home page. URL http://www.netbsd.org.

[155] NSLU2 Linux. The openslug linux distribution. URL http://www.nslu2-linux.org/wiki/OpenSlug/HomePage.

[156] F. Oldewurtel, J. Riihijarvi, K. Rerkrai, and P. Mahonen. The RUNES architecture for reconfigurable embedded and sensor networks. In *Sensor Technologies and Applications, 2009. SENSORCOMM '09. Third International Conference on*, pages 109 –116, 2009. doi: 10.1109/SENSORCOMM.2009.26.

[157] R. S. Oliver, I. Shcherbakov, and G. Fohler. An efficient operating system abstraction layer for portable applications in the domain of wireless sensor networks. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems*, SenSys '09, pages 379–380, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-519-2. doi: http://doi.acm.org/10.1145/1644038.1644111.

[158] OpenWSN Project. URL http://openwsn.berkeley.edu/.

[159] OSEK. OSEK/VDX operating system specification, version 2.2.3. URL http://portal.osek-vdx.org/files/pdf/specs/os223.pdf.

[160] J. Paek, B. Greenstein, O. Gnawali, K.-Y. Jang, A. Joki, M. Vieira, J. Hicks, D. Estrin, R. Govindan, and E. Kohler. The tenet architecture for tiered sensor networks. *ACM Transactions on Sensor Networks*, 6(4):1–44, 2010. ISSN 1550-4859. doi: http://doi.acm.org/10.1145/1777406.1777413.

[161] J. A. Paradiso and T. Starner. Energy scavenging for mobile and wireless electronics. *IEEE Pervasive Computing*, 4(1):18–27, 2005. ISSN 1536-1268. doi: http://dx.doi.org/10.1109/MPRV.2005.9.

[162] J. Polastre, J. Hill, and D. Culler. Versatile low power media access for wireless sensor networks. In *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2004.

[163] J. Polastre, J. Hui, P. Levis, J. Zhao, D. Culler, S. Shenker, and I. Stoica. A unifying link abstraction for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, New York, NY, USA, 2005. ISBN 1-59593-054-X. doi: http://doi.acm.org/10.1145/1098918.1098928.

[164] J. Polastre, R. Szewczyk, and D. Culler. Telos: enabling ultra-low power wireless research. In *Information Processing in Sensor Networks, 2005. IPSN 2005. Fourth International Symposium on*, pages 364–369, Apr. 2005. doi: 10.1109/IPSN.2005.1440950.

[165] R. Popescu-Zeletin, V. Tschammer, and M. Tschichholz. Y distributed application platform. *Comput. Commun.*, 14:366–374, August 1991. ISSN 0140-3664. doi: 10.1016/0140-3664(91)90062-6.

[166] A.-S. Porret, T. Melly, C. C. Enz, and E. A. Vittoz. A low-power low-voltage transceiver architecture suitable for wireless distributed sensors network. In *IEEE International Symposium on Circuits and Systems (ISCAS 2000)*, volume I, pages 56–58, May 2000.

[167] W. Pree. Meta Patterns - A means for capturing the essentials of reusable object-oriented design. In *Proceedings of the 8th European Conference on Object-Oriented Programming*, ECOOP '94, pages 150–162, London, UK, 1994. Springer-Verlag. ISBN 3-540-58202-9.

[168] F. M. Proctor and W. P. Shackleford. Real-time operating system timing jitter and its impact on motor control. *Sensors and Controls for Intelligent Manufacturing II*, 4563(1): 10–16, 2001. doi: 10.1117/12.452653. URL http://link.aip.org/link/?PSI/4563/10/1.

[169] ProtoGENI. ProtoGENI. URL http://www.protogeni.net/.

[170] D. Puccinelli and M. Haenggi. Reliable data delivery in large-scale low-power sensor networks. *ACM Trans. Sen. Netw.*, 6:28:1–28:41, July 2010. ISSN 1550-4859. doi: http://doi.acm.org/10.1145/1777406.1777407.

[171] PXA27x. Intel PXA27x processor family - design guide. Technical Report 280001-002, Intel, May 2005.

[172] V. Raghunathan, C. Schurgers, S. Park, and M. Srivastava. Energy-aware wireless microsensor networks. *Signal Processing Magazine, IEEE*, 19(2):40–50, Mar 2002. ISSN 1053-5888. doi: 10.1109/79.985679.

[173] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the 2001 conference on applications, technologies, architectures, and protocols for computer communications*, pages 161–172. ACM Press, 2001. ISBN 1-58113-411-8. doi: http://doi.acm.org/10.1145/383059.383072.

[174] S. Ratnasamy, M. Handley, R. Karp, and S. Shenker. Topologically-aware overlay construction and server selection. In *Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies, INFOCOM 2002*, volume 3, pages 1190–1199. IEEE, 2002.

[175] S. Ratnasamy, B. Karp, L. Yin, F. Yu, D. Estrin, R. Govindan, and S. Shenker. Ght: a geographic hash table for data-centric storage. In *Proceedings of the first ACM international workshop on Wireless sensor networks and applications*, pages 78–87. ACM Press, 2002. ISBN 1-58113-589-0. doi: http://doi.acm.org/10.1145/570738.570750.

[176] J. Rodrigues and P. Neves. A survey on IP-based wireless sensor network solutions. *International Journal of Communication Systems*. ISSN 1099-1131.

[177] K. Römer and F. Mattern. The design space of wireless sensor networks. *Wireless Communications, IEEE*, 11(6):54–61, Dec. 2004. ISSN 1536-1284. doi: 10.1109/MWC.2004.1368897.

[178] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *IFIP/ACM International Conference on Distributed Systems Platforms (Middleware)*, pages 329–350, Nov. 2001.

[179] RRDtool. URL http://oss.oetiker.ch/rrdtool/.

[180] D. Sanderson. *Programming Google App Engine*. O'Reilly Series. O'Reilly, 2009. ISBN 9780596522728.

[181] T. Schmid, H. Dubois-Ferriere, and M. Vetterli. Sensorscope: Experiences with a wireless building monitoring sensor network. In *Proceedings of the Workshop on Real-World Wireless Sensor Networks (REALWSN'05)*, Stockholm, Sweden, June 2005.

[182] F. Schmidt and W. Heller. Radio sensors powered by ambient energy: From strange ideas to mass market products. *Technisches Messen*, 74:12, 2007.

[183] M. Sgroi, A. Wolisz, A. Sangiovanni-Vincentelli, and J. Rabaey. A service-based universal application interface for ad hoc wireless sensor and actuator networks. *Ambient Intelligence*, pages 149–172, 2005.

[184] C. Sharp, M. Turon, and D. Gay. TEP102: timers. URL http://www.tinyos.net/tinyos-2.1.0/doc/html/tep102.html.

[185] S. Shenker, S. Ratnasamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. *ACM SIGCOMM Computer Communication Review*, 33(1):137–142, 2003. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/774763.774785.

[186] V. Shnayder, M. Hempstead, B.-r. Chen, G. W. Allen, and M. Welsh. Simulating the power consumption of large-scale sensor network applications. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 188–200, New York, NY, USA, 2004. ACM. ISBN 1-58113-879-2. doi: http://doi.acm.org/10.1145/1031495.1031518.

[187] J. Siegel. *CORBA 3 Fundamentals and Programming with Cdrom*. John Wiley & Sons, Inc., 1999.

[188] E. Souto, G. Guimares, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, and J. Kelner. Mires: a publish/subscribe middleware for sensor networks. *Personal Ubiquitous Comput.*, 10(1), 2005. ISSN 1617-4909. doi: http://dx.doi.org/10.1007/s00779-005-0038-3.

[189] W. R. Stevens. *UNIX Network Programming: Networking APIs: Sockets and XTI*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997. ISBN 013490012X.

[190] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160. ACM Press, 2001. ISBN 1-58113-411-8. doi: http://doi.acm.org/10.1145/383059.383071.

[191] C. Szyperski, D. Gruntz, and S. Murer. *Component software: beyond object-oriented programming*. Component software series. Addison-Wesley, 2002. ISBN 9780201745726.

[192] A. Tavakoli, P. Dutta, J. Jeong, S. Kim, J. Ortiz, D. Culler, P. Levis, and S. Shenker. A modular sensornet architecture: past, present, and future directions. *SIGBED Rev.*, 4 (3):49–54, 2007. doi: http://doi.acm.org/10.1145/1317103.1317112.

[193] D. Tennenhouse. Proactive computing. *Commun. ACM*, 43(5):43–50, 2000. ISSN 0001-0782. doi: http://doi.acm.org/10.1145/332833.332837.

[194] Texas Instruments. *MSP430F15x, MSP430F16x, MSP430F161x Mixed Signal Microcontroller Datasheet*. Texas Instruments, 2002. URL http://focus.ti.com/lit/ds/symlink/msp430f1611.pdf.

[195] The WUSTL Wireless Sensor Network Testbed. URL http://www.cse.wustl.edu/wsn/index.php?title=Testbed.

[196] J. R. Thorpe. A machine-independent DMA framework for NetBSD. In *Proceedings of USENIX Conference (FREENIX track)*. USENIX Association, 1998.

[197] TinyOS 2.x Core Working Group. Tinyos 2.x core working group. URL http://www.tinyos.net/scoop/special/working_group_tinyos_2-0.

[198] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, Istanbul, Turkey, Feb. 2005.

[199] A. Tomasic, C. Garrod, and K. Popendorf. Symmetric publish / subscribe via constraint publication. In *Proceedings of the First International Workshop on Performance and Evaluation of Data Management Systems (ExpDB 2006)*, 2006.

[200] Twisted Project. URL http://twistedmatrix.com/trac/wiki/TwistedProject.

[201] UC Berkeley. Omega testbed at UC Berkeley, telos (revision b) 802.15.4 wireless sensor network. URL http://omega.cs.berkeley.edu/.

[202] O. S. University. Kansei: Sensor testbed for at-scale experiments. In *Poster, 2nd International TinyOS Technology Exchange*, Feb. 2005.

[203] R. van Ommering. Building product populations with software components. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 255–265, New York, NY, USA, 2002. ACM. ISBN 1-58113-472-X. doi: http://doi.acm.org/10.1145/581339.581373.

[204] M. Vieira, J. Coelho, C.N., J. da Silva, D.C., and J. da Mata. Survey on wireless sensor network devices. In *Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA '03. IEEE Conference*, volume 1, pages 537 – 544 vol.1, sept. 2003. doi: 10.1109/ETFA.2003.1247753.

[205] S. Vinoski. Advanced message queuing protocol. *Internet Computing, IEEE*, 10(6):87–89, Nov. 2006. ISSN 1089-7801. doi: 10.1109/MIC.2006.116.

[206] P. Viscarola and W. Mason. *Windows NT device driver development*. New Riders Publishing Thousand Oaks, CA, USA, 1998.

[207] M. Waldvogel and R. Rinaldi. Efficient topology-aware overlay network. *ACM SIGCOMM Computer Communication Review*, 33(1):101–106, 2003. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/774763.774779.

[208] B. Warneke, M. Last, B. Liebowitz, and K. Pister. Smart dust: communicating with a cubic-millimeter computer. *IEEE Computer Magazine*, 34(1):44 –51, jan. 2001. ISSN 0018-9162. doi: 10.1109/2.895117.

[209] J. Waterman, G. W. Challen, and M. Walsh. Peloton: Coordinated resource management for sensor networks. In *HOTOS'09: Proceedings of the 12th conference on Hot Topics in Operating Systems*, Berkeley, CA, USA, 2009. USENIX Association.

[210] T. Watteyne. Hardware/software platforms. URL http://openwsn.berkeley.edu/wiki/OpenHardwareSoftware#HardwareSoftware.

[211] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, 2005.

[212] G. Werner-Allen, S. Dawson-Haggerty, and M. Welsh. Lance: optimizing high-resolution signal collection in wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 169–182, New York, NY, USA, 2008. ACM. ISBN 978-1-59593-990-6. doi: http://doi.acm.org/10.1145/1460412.1460430.

[213] D. A. Wheeler. Counting source lines of code (SLOC). URL http://www.dwheeler.com/sloc.

[214] K. Whitehouse and D. Culler. Calibration as parameter estimation in sensor networks. In *ACM International Workshop on Wireless Sensor Networks and Applications (WSNA'02)*, Atlanta, GA, USA, Sept. 2002. URL citeseer.nj.nec.com/mainwaring02wireless.html.

[215] K. Whitehouse, G. Tolle, J. Taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: using RPC for interactive development and debugging of wireless embedded networks. In *IPSN '06: Proceedings of the 5th international conference on Information processing in sensor networks*, pages 416–423, New York, NY, USA, 2006. ACM. ISBN 1-59593-334-4. doi: http://doi.acm.org/10.1145/1127777.1127840.

[216] Wireless Ad Hoc Sensor And Actuator Nets Lab, EFPL. Wasal testbed. URL http://http://wasal.epfl.ch/.

[217] WirelessHART Technology. URL http://www.hartcomm.org/protocol/wihart/wireless_technology.html.

[218] Wisebed. URL http://www.wisebed.eu/.

[219] G. Yang and M. Yacoub. *Body sensor networks*. Springer-Verlag New York Inc, 2006. ISBN 1846282721.

[220] X. Yang, N. Cooprider, and J. Regehr. Eliminating the call stack to save RAM. In *LCTES '09: Proceedings of the 2009 ACM SIGPLAN/SIGBED conference on Languages, compilers, and tools for embedded systems*, pages 60–69, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-356-3. doi: http://doi.acm.org/10.1145/1542452.1542461.

[221] I. Yannopoulos and D. Gay. TEP3: coding standard. URL http://www.tinyos.net/tinyos-2.1.0/doc/html/tep3.html.

[222] Y. Yao and J. Gehrke. Query processing for sensor networks. In *First Biennial Conference on Innovative Data Systems Research(CIDR 2003)*, Jan. 2003.

[223] ZeroMQ. URL http://www.zeromq.org/.

[224] B. Y. Zhao, J. D. Kubiatowicz, and A. D. Joseph. Tapestry: a fault-tolerant wide-area application infrastructure. *ACM SIGCOMM Computer Communication Review*, 32(1): 81–81, 2002. ISSN 0146-4833. doi: http://doi.acm.org/10.1145/510726.510755.

[225] ZigBee Alliance. URL http://www.zigbee.org.

Colophon

This document has been typeset using the LaTeX typesetting system created by Leslie Lamport and the *Memoir* class by Peter Wilson. The body text font is TeX Gyre Pagella, a clone of the Palatino font designed by Hermann Zapf.