Multitasking on Wireless Sensor Networks

Marcin Krzysztof Szczodrak

Submitted in partial fulfillment of the
requirements for the degree
of Doctor of Philosophy
in the Graduate School of Arts and Sciences

COLUMBIA UNIVERSITY

2015

ABSTRACT

Multitasking on Wireless Sensor Networks

Marcin Krzysztof Szczodrak

A Wireless Sensor Network (WSN) is a loose interconnection among distributed embedded devices called motes. Motes have constrained sensing, computing, and communicating resources and operate for a long period of time on a small energy supply. Although envisioned as a platform for facilitating and inspiring a new spectrum of applications, after a decade of research the WSN is limited to collecting data and sporadically updating system parameters. Programming other applications, including those that have real-time constraints, or designing WSNs operating with multiple applications requires enhanced system architectures, new abstractions, and design methodologies.

This dissertation introduces a system design methodology for multitasking on WSNs. It allows programmers to create an abstraction of a single, integrated system running with multiple tasks. Every task has a dedicated protocol stack. Thus, different tasks can have different computation logics and operate with different communication protocols. This facilitates execution of heterogeneous applications on the same WSN and allows programmers to implement system services. The services that have been implemented provide energy-monitoring, tasks scheduling, and communication between the tasks.

The experimental section evaluates implementations of the WSN software designed with the presented methodology. A new set of tools for testbed deployments is introduced and used to test examples of WSNs running with applications interacting with the physical world. Using remote testbeds with over 100 motes, the results show the feasibility of the proposed methodology in constructing a robust and scalable WSN system abstraction, which can improve the run-time performance of applications such as data collection and point-to-point streaming.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

The WSN technology comes from a vision of smart ubiquitous computing systems, indistinctly operating within our living environment. In 1965, in the science fiction book "Cyberiada", Stanisław Lem wrote about a civilization living on a desert, where the grains of sand are tiny computers that give the desert a superpower [120]. The science fiction story became a research quest when the *Smart Dust* term was introduced by Kristofer Pister in his DARPA proposal in 1997 [161]. The first prototype was delivered in 2002 [197]: it run with a 3MHz oscillator, was powered by a solar cell and used bidirectional optical communication. In parallel to the hardware efforts, David Culler's team started to work on TinyOS [80], a lightweight operating system to simplify the organization of a group of small sensors into a wireless mesh network. The first example of a practical WSN application was environment monitoring: sensors that used to be connected through wires became simpler and cheaper to install with the wireless technology.

The WSN shares hardware and software challenges with other related technologies. First, in 2006, NSF started to investigate distributed real-time systems with sensors and actuators operating on embedded devices and forming Cyber-Physical Systems (CPS) [115]. By closing the sensing loop, the CPS promised to collect data, process it and act on it, all without human intervention. While many CPS examples are less constrained in terms of power and computation, especially in the areas of autonomous robotics, they share similar software-development challenges with WSNs. For exam-

ple, Edward Lee [116] noticed that existing system design methodologies lack sufficient support for parallel programming and for expressing the notion of time. Second, on the business side, there is a growing interest in intelligent embedded devices, especially in the areas of smart homes, buildings and cities. The industry answer to this demand is the Internet-of-Things (IoT), new products that are enhanced versions of the existing embedded technology with the connection to the Internet. The IoT shares many hardware challenges with the WSN, including limited computation, communication and power resources. However, the IoT simplifies the application design and programmability by relying on the emerged cloud computing infrastructure to store, process, and transmit data [19; 26].

WSNs collect small amount of data representing sensor measurements [10; 32; 115]. Researchers have envisioned a wide spectrum of applications areas for WSNs, including construction [77; 101; 177], transit [12; 22; 114], energy [51; 58; 138], military [118; 166; 174], environment [84; 135; 185], and health-care [82; 106; 210]. Once an application is programmed and installed, WSN runs only this particular application, unless the whole firmware is reprogrammed either manually or remotely through a code that is integrated with the application firmware [87].

There are two main aspects that distinguish the WSN from the most common embedded systems, such as vending machines, a microwave, or a remote controller. First, wireless and often multi-hop networking is the essential part of any WSN application. WSNs establish ad-hoc communication by either forming a mesh network or a simpler star topology that has a one-hop distance to a more powerful device often connected to a power source. In either case, the application logic is supported by a network protocol that provides end-to-end communication. Second, the limited energy resources force the WSN to aim at the most power-efficient execution of the programmed application. While in many embedded systems power source is available from the grid or batteries replacement, in the WSN energy saving and operation lifetime is not a matter of convenience but existence. Since there is no demand for WSNs that are expensive to maintain or die shortly after the installation, the firmware running on them includes an energy saving protocol that usually turns off any unused microprocessor peripherals, especially the radio.

The WSN has been always defined as a smart system, driven by novel applications and self-manageable operation. In 2004, Feng Zhao from Microsoft Corporation and Leonidas Guibas from Stanford University published a WSN introductory book [211], in which they presented case studies

with WSNs aggregating measurements, clustering, time synchronizing, and tracking objects. In 2007, Holger Karl and Andreas Willig surveyed in their book almost 950 research publications with different networking protocols addressing various WSN application challenges [96]. In 2010, Jean-Philippe Vasseur from Cisco Systems and Adam Dunkels, author of the Contiki operating system, presented in their book a list of advanced WSN-based applications from the domains of smart grid, industrial automation, smart cities, home control, and others [194]. The reason why these applications have not been realized yet is because there continues to be a gap between the envisioned WSN functionalities and the existing software infrastructure to realize them [123].

1.1 Limited Resources, Restricted Design Methodology

There are hardware and software drawbacks that have prevented the WSN technology from becoming an intelligent, sustainable and autonomous system. The hardware on which WSNs operate is characterized by limited computation, communication and memory resources. These constraints are dictated by low-power operation for a long period of time. Due to the restricted battery capacity and size, the hardware must run on very small energy reserves.

The WSN software evolved from the embedded programming paradigm, where the system software consists of a single bare-bone thread of computation that operates continuously. Despite improving the software by providing hardware abstractions and simplifying the network communication, so far the practical functionality of the WSN has been limited to a single application. Out of the box, the WSN software only supports low-power data collection.

One of the WSN programming challenges is the applications distributed nature. A programmer faces independently operating hardware units processing data. Due to wireless communication, the data is delayed and sometimes lost. The programmed logic keeps track of a single device's state, but there is no notion of the WSN state, a state that would be consistent across all the devices in the network. Thus, the WSN is programmed from an individual network-device perspective, not as a unified and integrated system. This programming approach complicates expressing a network task as a logic that asynchronously executes in parallel on multiple independent devices, loosely connected through an unreliable network.

This dissertation is motivated by the observation that after a decade of research it contin-

ues to be difficult to program other applications than the simple data collection. There are no methodologies or system design principles to support multitasking in WSNs. This has multiple consequences. First, it becomes difficult to program robust applications. Even the simple data collection application still requires additional logic supporting the WSN installation and deciding what to do during system failures or reboots. Second, it becomes difficult to program complex applications. For example, tracking applications often require network synchronization. The complete tracking application logic can be programmed as a single task or it can be split into two tasks, one for tracking and one for synchronizing. The latter would allow programmers to apply modular system design, which simplifies programming and promotes code reuse. Finally, the existing software does not support the execution of multiple different applications across the same WSN without reprogramming the whole embedded firmware [70; 87; 130; 208]. This results in different applications or different products operating on separate WSN installations, thus leading to hardware redundancy and increased installation and maintenance costs.

1.2 Multitasking on Wireless Sensor Networks

To address these challenges I propose a system design methodology to divide and conquer the hardware resources and applications logic challenges into smaller tasks and embrace the multitasking notion of operation across the whole network. My argument has two main reasons for the WSN design methodology that:

- **supports multiple applications** to address three WSN challenges: (1) executing more than one application per WSN installation, (2) executing both heterogeneous and similar applications with different run-time constraints, and (3) scaling application complexity over a set of multiple smaller tasks.
- **distinguishes between the application and the system logic** to introduce three WSN system design abstractions that: (1) define the relation between the WSN applications and the WSN system, (2) provide system services enabling multitasking, and (3) allow stand-alone system services responsible for controlling the WSN run-time operation.

Multitasking in WSNs raises fundamental questions. What are the tasks? What and where do the tasks execute? What are the system tasks and how do they relate to the application

tasks? How are the tasks scheduled and how do the tasks communicate? Finally, how can these tasks be implemented and used across the WSN? All these questions emerge at the network-level scale, assuming that the WSN operates as a unified system. However, the current software development practices offer programming of the individual devices and rarely see the WSN as an integrated system consolidating resources of all the participants in the network into a single computing platform.

The following is a description of the proposed system design methodology for multitasking on WSNs.

Building Block: A Task. The WSN executes units of work that are called tasks. These tasks operate across the whole WSN, not on any specific network device. This system design approach fundamentally differs from the previous WSN architectures, because instead of fixing the software logic for a specific application or fixing communication across the network, the system tracks the execution of the tasks, each comprised of its own computation and communication logic.

Organizing Tasks: At Design-Time and At Run-Time. A task consists of parallel computation that is physically distributed and loosely connected. Because a WSN work cannot be accomplished without communication among multiple devices, a task comprises of computation and communication logic integrated and executing together. In software, a task is distributed across a layered stack architecture, where every layer has a specific role, such as executing local computation, establishing communication, and controlling low-level hardware performance. At run-time, every task runs on its own stack, with its own computation and communication logic that can differ from other tasks that run concurrently or are scheduled at other times.

Figure 1.1 shows the execution model of a WSN designed with the proposed methodology from the software implementation perspective. The system is designed as a network protocol stack, with each layer consisting of multiple implementations of the same functionality, but with logic addressing different objectives. At run-time, a task executes one instance of a service implementation from every layer of the stack, choosing computation and communication services that meet a given task requirements. An identical copy of the stack executes concurrently on every device that is part of the network. In the figure, the arrows between the network protocol stacks represent the WSN context switch among the tasks. When the WSN switches a task it changes the computation and communication logic running on every device.

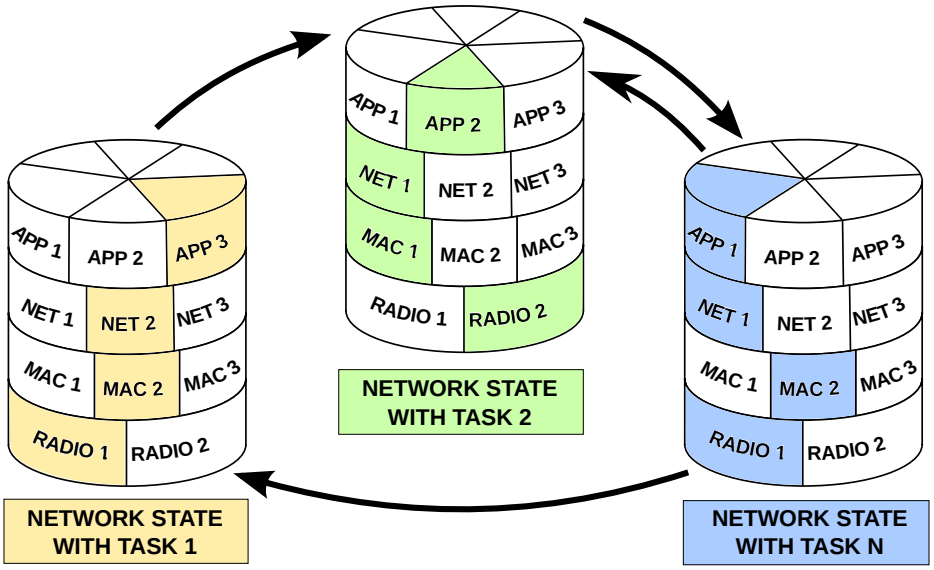


Figure 1.1: Run-time context switch among the three tasks running across the WSN.

Types of Tasks: Application and System. Tasks can perform applications and system-level works. An application may consists of a stand-alone task. For example, data collection can run as a single task, with computation logic sampling sensors and communication logic transferring the sensor samples toward a destination, such as a computer or a gateway to another network. An application may also consists of multiple tasks executing concurrently or consecutively. A task can also execute a system-level work. For example, a task can monitor the energy left on the low-power devices. Thus, a task can perform a maintenance job. A system task can also provide services and abstractions simplifying applications programming.

Generalize Toward System Design. The notion of an organized distributed computation and communication can be abstracted to a notion of tasks operating on a single, integrated distributed system. This system design abstraction is shown in Figure 1.2. Here, both the system tasks and the application tasks are not operating directly on the network devices but on a fabric that integrates all the hardware resources into a single system. Such software integration requires system-level abstractions to hide the limitations and the run-time overhead of the physically distributed hardware units.

Tasks Scheduling. A behavior of a task starting on a network differs from that of a task starting on a single device. Scheduling a new task across the WSN causes an unequal execution

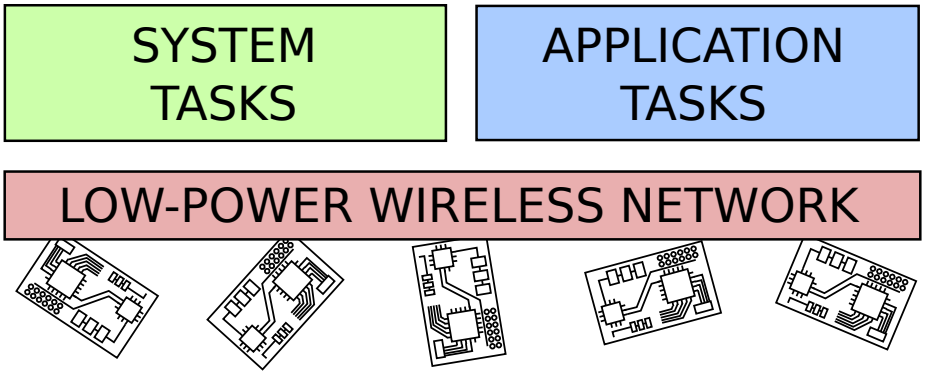


Figure 1.2: A single, unified system abstraction executing system and application tasks.

initiation latency and an unsynchronized parallel operation. To provide the abstraction of a single integrated system, a special WSN system service minimizes the distributed implementation overhead and provides a synchronized scheduling of a task on all the network devices.

Tasks Communication. While the tasks running on the same device can communicate through a shared memory, a computation executing across a network requires a dedicated protocol to establish communication channels among computation entities. To hide the distributed nature, the WSN system provides a shared memory that is synchronized on all the network devices. This allows multiple tasks to communicate without the need to specify the physical location or the specific network device from where a task can initiate a communication with another task.

The thesis presented here is that *the execution of applications and system-level services as multiple heterogeneous tasks composed of its own computation and communication logic is the key to create an abstraction of a single, integrated system, running across the low-power wireless sensor network devices.*

1.3 Related Work

The WSN research focuses mainly on the networking aspects and the issues related to the energy consumption, and not on the WSN programming and the models of computations that would drive this technology beyond labs' prototypes. Surprisingly, in the WSN it is the software that fails to advance along research contributions and scale with the applications and the system complexity [52; 31]. Consequently, even the computer industry finds the existing WSN software development

approach difficult to maintain, modify and reuse across the projects [15].

The existing infrastructure for development of the WSN software includes tools and bare-metal operating system primitives running on microcontrollers and small microprocessors. TinyOS [127] and Contiki [41] are two leading academic examples of embedded operating systems. Both systems come with a set of radio drivers and networking protocols that can be reused by the application program running on the devices supported by one of these systems. FreeRTOS is used in many products on the market, from toys to aircraft [59], but it does not include WSN protocol libraries.

Distributed embedded devices need to be organized in a form of a unified network architecture to coordinate a task among themselves [52]. The common approach is to apply the layered system design, separating the low-level networking service from the application logic [31]. A natural candidate is the IPv6 stack, especially its compressed version known as 6LoWPAN [86], which is standardized by IETF and added into TinyOS and Contiki [104], as well as an independent library for FreeRTOS. To further hide the distributed nature of the WSN, IETF standardized IP-based Constrained Application Protocol (CoAP) [107; 171] that provides abstraction for connecting WSN with an outside application. CoAP allows to request and retrieve sensor data through REpresentational State Transfer (REST) Web architecture [56; 34] operating within HTTP. Another example is TinyDB [140], which creates a data-base abstraction for the WSN such that sensors' measurements can be retrieved with an SQL-like program, without specifying any communication protocol. While HTML and SQL syntaxes are familiar to many potential WSN programmers, they can only support the simplest WSN applications, such as sensor data retrieval. In these programming abstraction examples, the cost of hiding the distributed system nature comes with limited WSN functionality.

In the WSN the outcome of a distributed computation, both in terms of computation precision and resource overhead, relies on the form of communication, especially the MAC and the network protocols. Realizing how critical is the role of communication in WSN applications, some researchers advocate a cross-layer software design [134; 144; 148; 150], where the application logic flow can be used to adapt the network communication. The cross-layer approach promises run-time WSN performance optimizations, but it does not lead to a scalable methodology for system design. It also abandons the Open System Interconnection model [216] and therefore complicates programming and maintenance of a distributed embedded software.

One way of hiding the distributed nature of the WSN without compromising the communication

performance is to constrain the need for data delivery, especially by avoiding multi-hop and mesh radio transmissions among the low-power devices. Tenet [157] is a system with its own programming syntax that allows programmers to express computation on the WSN. It supports a tiered network architecture, assuming that among the low-power embedded devices there are some more powerful ones that can organize computation on the network and establish multi-hop communication with other more powerful units. Tenet aims to address the most common WSN deployment scenario, where every low-power embedded device is a hop away from a more powerful one, which is always ready to receive a communication [50; 214]. However, besides fixing the network architecture, Tenet does not allow different tasks to run on their own communication protocols.

Another example of orchestrating computation and communication across a WSN is a bottom-up approach, which first addresses the low-power multi-hop radio challenges, and then builds an integrated system and services on top of the defined radio-level primitives. Glossy [55] provides fast data flooding and precise time synchronization on the WSN. It exploits constructive interference among multiple, synchronized and identical radio packet transmissions [47; 45]. This very efficient data dissemination technique can be used to establish a data bus [54] and perform simple computations across the WSN [112; 201]. Glossy and the services build on top of it integrate all the network devices into a single system. However, this approach has some practical limitations. For example, Glossy implementation puts run-time software constraints on the embedded system. While future more powerful hardware architectures and new radios may solve this limitation, Glossy, as well as other optimized data flooding techniques [139; 36] do not follow the standard MAC protocols, and thus are difficult to integrate with the existing IEEE 802.15.4 systems. Further, theoretical works show that constructive interference may not scale very well [196].

This dissertation addresses the challenges in scaling out the design and functionality of the WSN by supporting the execution of multiple applications and providing network-wide system services. My work introduces algorithms and protocols, necessary to create an abstraction of an integrated and consolidated system. To support WSN macro-programming, I define a new programming language. The proposed WSN software architecture embraces modular and layered system design, and allows run-time communication optimizations for specific applications.

1.4 Outline of the Dissertation

This dissertation is organized as follows:

In Chapter 2, I describe the WSN background, including state of the art in the WSN system design and the WSN challenges that motivate my work. I describe the existing hardware architectures used in the WSNs. The limited hardware resources impose run-time constraints, which drive my research as well as other WSN-related works to look beyond the existing solutions. Then, I discuss the software development frameworks and tools used to check the embedded firmware fidelity and to measure its performance. Finally, I present various approaches to program the WSN data collection application, and I summarize related works on one of the most important WSN system metrics: the low-power operation.

In Chapter 3, I address the problem of running multiple heterogeneous applications on a WSN. In the context of WSN system design, this chapter investigates the relation between computation and communication, showing that different network computations require different communication mechanisms. I present and analyze the problem of scheduling and supporting the execution of multiple heterogeneous applications on top of the same WSN. First, I establish that using the same MAC or network protocol is not sufficient to obtain acceptable performance across a set of applications that require different types of communication services from the protocol stack (e.g., low-rate reliable many-to-one collection vs point-to-point low-latency bulk-data streaming). Hence, I propose a framework, called Fennec Fox, to dynamically reconfigure the WSN and adapt its power consumption, transmission reliability, and data throughput to the different requirements of the applications. The framework makes it possible to specify, at design-time, distinct network, MAC and radio protocols for each application as well as the events and policies triggering the WSN reconfigurations. At run-time, the WSN automatically reconfigures itself in response to these events and according to these policies. Through experiments on a 119-node testbed, I show that the proposed approach can reconfigure the whole network in few hundreds of milliseconds while incurring little memory and control overhead.

In Chapter 4, I present a WSN running with applications and system services that are crucial for sustainable WSN installation. I describe the modeling, implementation, and evaluation of a single WSN that executes energy-harvesting algorithms and sensing applications. The network energy-management is modeled as a feedback control system. An asynchronous execution of the

energy-management and the application processes is modeled as a finite state machine. To evaluate this approach, I design energy neutral sensing systems for two applications and implement them with the framework introduced in Chapter 3.

To test and evaluate a WSN application interacting with the physical world, one needs to deploy a testbed. In Chapter 5, I introduce an open framework for an efficient deployment of heterogeneous wireless testbeds for Cyber-Physical Systems (CPS). The testbed architecture, which can be configured and optimized for each particular deployment, consists of a low-power wireless network of embedded devices, a backbone network, and a server back-end. The framework, whose source code is publicly available, includes a comprehensive set of software tools for deploying, testing, reconfiguring, and evaluating the CPS application software and the supporting firmware. I then discuss the architecture, the framework properties, and the hardware resources that are necessary to deploy an experimental testbed. Next, I present two case studies built with the framework: an outdoor lighting installation in a commercial parking lot and an indoor university building instrumentation. Using the two deployments, I evaluate experiments normally conducted by CPS engineers to better understand the environment in which the CPS is deployed. The results of these experiments show the feasibility of the proposed framework in assisting CPS research and development.

Chapter 6 introduces the second generation of the WSN system, with computation and communication logic operating as tasks across the network. Chip and Dale are a distributed system and its programming language, which on the WSN create an abstraction of a unified and integrated system operating with a single, high-level program. The system executes tasks across the WSN. The tasks either run with application logic or implement system services. I present two system services that allow tasks to (1) communicate through a small memory shared across the system, and (2) start synchronously executing across the WSN. I define the system concepts and building blocks by providing Dale code examples. Using two remote testbeds, I evaluate the performance of the system services. I demonstrate Chip and Dale performance on a data collection application example, showing that the provided abstraction and the system services permit to synchronously duty cycle the whole network and to adjust at run-time the application sensing rate. For data collection period set to 15 and 30 minutes, my design approach improves duty cycle by 90% and saves power by 60% with respect to the default WSN programming techniques using the same

application, network and MAC protocols.

I conclude with Chapter 7 that summarizes the main contributions of this dissertation and describes cases where some of these contributions were used in collaborative research projects. The developed software, which is open-source and publicly available, was used in a multi-disciplinary academic research project EnHANTs. It was also applied by researchers in industry at the Philips Research North America and the United Technologies Research Center. I finish this chapter by outlining the most promising avenues for future research. I propose to further enhance the run-time tasks scheduling, which would enable addition of new tasks to the system without reprogramming the whole firmware. Another direction of future research is the integration of the standardized IP-based protocols with the existing software framework to provide interoperability with industrial systems.

Terminology: task vs. process. In Computer Science, words *task* and *process* are often used interchangeably. In this dissertation, *task* is used in the context of the system design methodology for multitasking on WSNs. However, when examples of the WSN software implementations are presented, especially in the sections referring to the high-level programming language for WSNs, *process* is used to denote an instance of a code that is executed across the network. The only distinction in the use of these two words is in Chapter 6, where *tasks* are *processes* that are scheduled by programmers, but not all *processes* are *tasks*.

Chapter 2

Background

This chapter describes the background of the dissertation. In the world of high-performance computing and big data it is sometimes hard to imagine technology where every computed or communicated bit counts, technology where MB of storage is a luxury and where Kilo- is the common prefix. This chapter explains the nature of the Wireless Sensors Network (WSN) that sacrificed Giga- and Tera- computation and communication performance for operation lifetime measured in years. The first part of this chapter describes WSN hardware and software. It outlines the WSN technology constraints, from the physical resources as well as the existing software components and programming approaches. The second part illustrates the typical use case of the WSN. It presents the main application of the WSN - the data collection, and characterizes the main system performance metric, which is the efficient low-power execution measured as the WSN duty cycle.

2.1 Motes from a Hardware Perspective

The WSN consists of tens to hundreds of tiny low-power wireless devices called *motes*. A typical mote is build with a tiny microcontroller running at few MHz, with memory on the order of KB, a low-power IEEE 802.15.4 complaint radio, few sensors connected through analog-to-digital converter (ADC), and a battery. The most popular motes are manufactured as research and development platforms. The platforms include processing unit, radio, and extension interfaces such as GPIO, ADC, I^2C, SPI, and UART, which are used to expand the platforms with more sensors and actuators, and to program the firmware into the motes ROM memory. Due to their small size

and the expected long term operation, WSN hardware is difficult to improve. Since no research breakthroughs have dramatically changed this technology paradigm, the current computation, communication, memory, and power constraints in the WSN development are likely to continue in the near future.

Computation. One of the first popular microcontrollers used in WSN was ATmega128L [11], Atmel 8-bit architecture with 128KB programmable memory and 4KB of ROM. With these parameters, this microcontroller is similar to Intel 8088 microprocessor from 1979 [37], running in the first IBM PC [89]. ATmega128L was used in one of the popular platforms from the Mica family [81], called MicaZ mote [146]. On MicaZ, ATmega128L run at 8MHz and with 2.7-5.5 operating voltage. The microcontroller consumed 8mA in the active mode and less than 15μA in the sleep mode.

One of the most important criteria in microcontroller evaluation is its power consumption in the active mode and the sleep mode. To ensure long operation, WSN motes are duty cycling, switching between the on-and-off cycles. A mote is in an active mode when it performs some work that requires powering the microcontroller and some other hardware peripheral. Usually, motes operate with less than few percent duty cycle, turning off for long periods. Therefore, the microcontrollers' sleep mode power consumption specification plays an important role in the design of new WSN platforms.

For example, a device can be powered by a single alkaline AA battery with 2800mAh capacity. This device running ATmega128L without duty cycling can operate for over 13 days. At 100% duty cycling, this device can stay in the sleep mode and do nothing for almost 20 years. At 1% duty cycle, for example starting a 1 second job every 100 seconds, ATmega128L consumes on average 0.09485mA and can work at this rate for over 3 years.

In actual WSN products, improving device lifetime requires more careful engineering. First, there are other hardware peripherals, such as sensors, ADC, and radio, which also need to be duty cycled, and they have their power consumption specifications for the active and sleep modes. Second, for every duty-cycled peripheral, there is a trade-off between the energy consumed in the sleep mode and the amount of time it takes to switch back to the active mode.

Since the power consumption impacts the WSN lifetime it is not surprising that the next popular microcontroller used in the WSN was more energy efficient. Texas Instruments introduced a successful family of low-power microcontrollers called MSP430. One example is MSP430F1611,

Figure 2.1: Tmote Sky also known as TelosB mote. The most popular mote used in the WSN research and development. Most of the experiments presented in this dissertation were conducted with this WSN platform.

a 16-bit microcontroller with 48KB of Flash and 10KB of RAM [189]. Running at 8MHz, it consumes 1.8mA in the active mode and 5.1μA in the sleep mode. MSP430F1611 owes its success to TmoteSky, also known as TelosB [163], which for years was the most popular WSN platform, and is still used in many research labs. The newer generation of this MSP430 microcontroller, MSP430F2617, comes with another popular WSN platform, a successor of TelosB, called Zolertia Z1 [217].

Figure 2.1 shows a picture of Tmote Sky. From left to right, the platform is built with an embedded antenna, the IEEE 802.15.4 complaint radio, and USB to serial port bridge, which is used to program the mote and to communicate with the firmware. The MSP430 microcontroller is located on the other side of the mote. When purchased from Memsic, this mote comes with a battery holder for 2xAA batteries and with integrated humidity, light and temperature sensors [147].

ARM architecture was one of the key game changers that led to the mobile and smart phone computing evolution [57]. The instruction set architecture adopted by all the major chip makers reduced the manufacturing costs and provided more energy-efficient operation. Although ARM is famous mostly due to its presence in smart phones, tablets, and entertainment electronics, ARM expanded its portfolio and now competes with microcontrollers such as ATmega128L and MSP430, and has became the academic [105] and industry favorite choice in the IoT products line.

The Cortex-M is a family of ARM 32-bit microcontrollers characterized by low-cost, small-size, and low-power operation [7]. These microcontrollers use the Thumb-2 instruction set architecture that produces high code density. They provide low interrupt latency, can run at few to tens of mA and support various sleep modes from few μA down to nA. Depending on the type of the sleep mode, the wake-up time can take between few tens and hundreds of μ-seconds.

There are four Cortex-M microcontrollers suitable for WSN and IoT systems operating with limited energy resources [7]. The smallest and the cheapest are M0 and M0+, which directly compete with the 8-bit and 16-bit architectures. An example of a WSN platform with Cortex-M0 is the Michigan Micro Mote [119], a $1mm^2$ stacked system with low-power I^2C bus [110]. Cortex-M3 is a microcontroller suitable for highly deterministic real-time applications. It is used in WSN platforms already sold for research and development. The first example is Lotus from Memsic [145]. Lotus runs on Cortex-M3 operating between 10-100MHz. It has 64KB of RAM and 512KB of programmable flash. In the active mode, Lotus consumes 50mA at 100MHz and 10μA in the sleep mode. The second example is Opal [94]. It has Cortex-M3 manufactured by Atmel and called SAM3U. This microprocessor runs up to 96MHz, with 52KB RAM and 256 ROM. Opal, used by the Commonwealth Scientific and Industrial Research Organization (CSIRO), consumes 8.9μA in the sleep mode. Finally, the fourth example is Cortex-M4. This microcontroller combines high-efficiency with low-power execution and low-cost. With memory protection unit (MPU), digital signal processor (DSP) and floating point (FP) support, Cortex-M4 is a very attractive microcontroller for smart appliances in the IoT domain. Together with the whole ecosystem, including rich documentation, large number of development-support tools as well as middleware and RTOS suites, ARM Cortex-M family of microcontrollers will continue to define the computation and memory standards of the hardware architectures for WSN.

Radio Communication. The WSN radio communication is predominantly defined by the IEEE 802.15.4 standard [90]. The IEEE 802 is a family of standards related to networking, where 802.15 is a special committee working on wireless personal area networks (WPAN). IEEE 802.15 is subdivided into multiple tasks groups, such as mesh networking, visible light communication, or body area networks. The most popular tasks groups include the 802.15.1, which provides foundations for the Bluetooth technology, and the 802.15.4, which standardizes the physical (PHY) and the medium access control (MAC) layers of the protocol stack used in the low data-rate and the

low-power wireless communication.

One of the most popular radio chips implementing the IEEE 802.15.4 standard is CC2420 [188]. It is adapted in many WSN platforms [3; 141; 146; 163; 217], including TelosB and Zolertia Z1. Operating at 2.4GHz band, the radio provides data-rate at 250kbits per second. Transmitting at 0dBm, the radio consumes 17.4mA and 18.8mA in the receive mode. In the idle mode, CC2420 runs on 426μA and requires 1μA when voltage regulator is turned off.

The IEEE 802.15.4 standard allows to use sub-1GHz and 2.4GHz frequency bands. WSN platforms, such as Opal [94] or UCMote Proton B [193], have two radios running on different frequency bands. Using multiple radios on the WSN platforms solves some wireless communication challenges [33], particularly it improves end-to-end delivery rates and network stability [111; 207].

Whereas WSN research concentrates on radio communication following the IEEE 802.15.4 standard, industry integrates other wireless communications in the low-power devices, particularly those entering the market as part of the IoT. One example is Bluetooth. Intel Mote [102] is a WSN platform using the original Bluetooth standard technology, which nowadays is replaced in the low-power devices by Bluetooth low energy (BLE), also known as Bluetooth Smart [16]. With BLE, the IoT devices can be controlled from a phone, which simplifies the design of new applications. Another example of the communication technology integrated with the IoT devices is IEEE 802.11, known as WiFi. At high data payloads, WiFi is more energy efficient than IEEE 802.15.4. Current WiFi radio chips support low-power sleep mode operation, consuming less than 1μA for the price of a longer wake-up time on the order of ms. Further, WiFi allows connecting the low-power devices with other existing electronic equipment, particularly the WiFi routers that enable Internet connectivity.

The WSN communication technology leans toward heterogeneous radios with different physical and access control standards. Low-power consumption requires the IEEE 802.15.4 standard. WiFi is necessary for the Internet connectivity. BLE simplifies interactions with the users through their phones. These wireless communication standards require different amounts of power and already operate on existing devices that can serve as a bridge among radio technologies.

Power. The design of the WSN platforms and other low-power wireless devices aims at fitting the required computation and communication components within a given power budget. Besides microcontrollers and radios, other application-specific hardware peripherals must be considered in

this context. For example, sensors may consume a significant amount of power, from tens of μA up to tens of mA per measurement. To operate all the hardware components, the WSN platforms consume energy from one or more sources:

- Battery: for tiny motes, button cells have 200mAh capacity. Most of the WSN platforms run on 2xAA batteries, each with ∼2600-3000mAh capacity.
- Energy-Harvesting: energy can be harvested from multiple sources [65; 73] but solar energy is the default choice in research and industrial products. As the devices scale down in size, solar cell can deliver more power than Lithium battery [206].

Recent improvements in the design of power efficient hardware promise to operate a WSN with a longer lifetime and achieve more with the same energy resources. In microcontrollers, power can be saved by using deep pipelines [168] and by dynamically adjusting the voltage threshold [6]. New sensors become more energy efficient [85; 131; 99], and better understanding of the sensor signal characteristics can also save energy in the analog-to-digital converters [204]. In the wireless radio, adaptation of pulsed ultra-wideband (UWB) communication can lead to further power savings [149; 195]. Duplex-radio communication [91] can increase wireless efficiency. In the future, WiFi backscattering [98] and further advancements in solar energy harvesting [205; 66] may eventually lead to battery free devices.

2.2 Motes and Their Software

The WSN motes are tiny computing devices operating with firmware that fits into their limited memory resources. The firmware is developed by combining the existing embedded operating system libraries with new application logic. At micro-scale, every mote can be programmed individually with a logic that includes communication mechanisms allowing devices to collaborate. At macro-scale, the whole WSN can be programmed at once with the application logic. In either case, the resulting software is tested and evaluated through WSN simulators and testbeds. A testbed consists of a network of a motes and a mechanism to reprogram all of them at once with a new firmware.

Operating System. Low-power wireless distributed embedded systems run with real-time operating systems (RTOS), not with Linux. Linux and other UNIX derivatives dominate across

other computing platforms, from smart phones through smart electronics and desktops up to cloud and high-performance systems. Two quantitative reasons why Linux is not a good match for the low-power wireless devices domain are:

1. ***Computation and memory resources***: One of the latest Cortex-M4 microcontroller examples, Freescale Kinetis KW2x, can run up to 50MHz, with maximum 512KB of programmable Flash and 64KB RAM memory, which are not sufficient for Linux. Further, Cortex-M4 does not have a memory management unit (MMU), which is required for Linux.

2. ***Booting Delay***: Low-power devices demand fast transition between the sleep and the active mode to minimize the duty cycle period. Whereas a standard Linux kernel needs seconds to start, RTOS systems start, run an application, and shutdown in a time on the order of milliseconds. Later in this dissertation, I describe a case in which a system wakes up, runs a fully-featured WSN application across the network of 100 devices spread across a 3-floor building and returns to sleep, all within few seconds.

The computation, memory, and energy constraints preclude using Linux on low-power devices without a stable source of power. However, Linux continues to be present in the WSN and IoT ecosystems, running on more powerful embedded devices, such as WiFi routers. These devices serve as a bridge between the energy-constrained devices and other computation platforms. They provide Internet connectivity and data storage, and they can be used as hubs interconnecting low-power sensors into a single system [157; 190].

The WSN motes run with embedded operating systems that include basic run-time services, which are extended with an application logic and compiled for a given platform architecture. Some of the embedded systems proposed by researchers for the WSN are MantisOS [14], SOS [71], Nano-RK [53], Nano-Qplus [159], RETOS [24], LiteOS [21], and Pixie-OS [137]. Each of these systems improves or solves a specific WSN issue, such as power consumption, system programmability, or software modular design. However, most of them are discontinued projects because they are not widely used.

The two major embedded systems that have survived the test of time and public acceptance are TinyOS [127] and Contiki [41]. Both of these operating systems support major WSN platforms and contain the most common network protocols, especially those providing data collection. Both

systems established a complete software development ecosystem, with their simulators, web-sites, and mailing lists. A decade after their public release, both TinyOS and Contiki continue to be used in research and development as well as in some IoT products.

Contiki is an open-source operating system for the Internet of Things. It supports the major IETF protocols for low-power networking, including RPL [200] and CoAP [171]. It is written in C and incorporates various memory allocation techniques and lightweight stackless threads programming model. Building software with Contiki is less complicated thanks to the active community of developers and a powerful WSN simulator with graphical user interface [156].

TinyOS is also an open-source operating system, designed primarily for research and development in the WSN. TinyOS is programmed in nesC [61], a C-dialect integrated with event-driven execution and stimulating modular design of the embedded applications. TinyOS runs with major WSN routing protocols, supports many WSN platforms and provides lightweight WSN simulator [126].

The work presented in this dissertation is based on TinyOS. TinyOS and nesC offer mechanisms to program every individual network device. One of the contributions of this dissertation is a software, which is based on TinyOS and installed on every mote to provide an abstraction of a single, integrated system. The system presented in the following chapters is programmed in nesC. Whereas nesC programs specify computation logic for each individual mote, this dissertation presents two generations of a language to program the whole WSN at once, on top of the introduced network-wide system abstraction. Programs written in this language are compiled into nesC that is merged with the rest of the code and further compiled again as a TinyOS application.

TinyOS programming combines existing nesC code with new application logic [125]. A TinyOS system is a set of components connected through interfaces. The components that come with the TinyOS source code include implementations of the network protocols, primitive system services, and drivers for platform peripherals, such as radio or ADC. An application is programmed as another component or a set of components. At compilation time, the components source code stored in multiple files is analyzed and optimized to minimize the chance of race conditions and memory footprint. The result of the compilation is a C file, which is further cross compiled into a mote binary for the architecture used by the specified WSN platform.

Figure 2.2 shows an example of a TinyOS module written in nesC. The name of the module

```
module DelayedCounter {
      provides interface Delay;
      uses interface Leds;
      uses interface Timer<TMilli>;
}

implementation {
      uint8_t c;

      command void Delay.add(uint32_t delay) {
            c++;
            call Timer.startOneShot(delay);
      }

      event void Timer.fired() {
            call Leds.set(c);
      }
}
```

Figure 2.2: An example of a TinyOS module written in nesC.

is *DelayedCounter*: it sums the number of function calls until no more calls are received for *delay* measured in milliseconds and then displays the computed sum on a platform LEDs. This module interacts with others through function calls, which are part of an interface. In this example, the module provides the interface *Delay*, which means that this module implements all the functions of this interface and other modules can call these functions. *DelayedCounter* uses interfaces *Leds* and *Timer*, which are functions implemented by other modules. Interfaces, such as *Timer*, can be parameterized, for example to specify the computation unit. In this case, the interface specifies the timer unit, which is millisecond. The interface functions implementation begins with keyword **command**. Using some interfaces may requires implementing functions called **events**. Implementations of these functions are called by the module that provides such interface.

Implementing a networking application in TinyOS requires connecting the interfaces provided by a routing protocol module with a module implementing an application logic. The protocols provide interface that consist of functions allowing to send a message, and interfaces that signal events when a message is received. With this programming approach, simple applications can quickly become complicated, especially when they need to keep track of their application code and the state of interaction with other modules. For example, a module may need to wait to send a

message until another message is received and a sensor reading meets a predefined threshold value. The application state-space is further complicated when the application logic differs according to the role the device it runs on plays in the WSN. For example, an application may perform different actions when it runs as a router and execute other logic when it runs on a platform with a specific type of a sensor. In all these cases, WSN programmers must be fully aware of the application's distributed nature and the potential run-time errors and bugs resulting from unsynchronized and unreliable wireless communication.

Macroprogramming. To simplify the WSN programming, researchers have proposed higher-level programming abstractions [28; 68; 109; 124; 140; 180]. The macroprogramming frameworks, such as sMapReduce [69], MacroLab [83], and Kairos [67], introduced new languages that permit to program the whole WSN as a single system. Unfortunately, since none of the previous works found wider public acceptance, these macro-programming techniques died out. One of the limitations of these works is that they focus only on the complexity of expressing a distributed computation and address it by simplifying communication or by using a single network protocol for all the types of distributed computations. Consequently, by simplifying programming, authors sacrificed either communication performance or the language's expressiveness.

This dissertation provides techniques for macro-programming of the whole WSN. One of the fundamental distinctions between this work and the previous studies is the relation between computation and communication. In this dissertation, every computation is combined with a dedicated communication mechanism. This approach allows us to use exactly the same protocols and applications as those from TinyOS, but it also offers methodology to scale the programmed firmware with more applications and systems services, which may need to use different communication mechanisms and different radios. The presented macro-programming is not used to simplify the individual applications or computations and their assigned communication protocols, since these are still written in nesC, but to simplify the scaling of the complexity of the WSN, and where possible, reducing the distributed system programming challenges by providing abstractions of a single, integrated computation platform.

Tools. The WSN has been a topic of academic research for over 15 years. It took a decade to address the low-level hardware/software challenges [76] and introduce reliable network protocols for data collection [30]. The development of the embedded software has accelerated with the

introduction of the WSN simulators, such as TOSSIM [126] and COOJA [156]. These simulators support development and testing of the embedded code in a software-simulated wireless network environment.

One of the simulator development limitations is the simplified model of wireless communication and hardware execution. Due to the network dynamics at the radio link layer [176], it is difficult to create a model of a wireless communication that would truly represent all the possible situations that can occur in an actual deployment [95; 117; 175]. Thus, an adequate testing of an application or a network protocol performance, requires implementing and running experiments on real hardware.

Testing WSN for robustness and scalability demands significant resources. For these reasons, universities around the globe build and contribute to the research community open-access and remotely-controlled testbeds [2; 9; 17; 35; 75; 78; 129; 132; 199]. These testbeds, which often consist of tens or hundreds of motes deployed across the offices and floors of academic buildings become de facto benchmarks that are used to verify and evaluate the WSN software. Testbeds usually provide a web-based interface to upload a firmware and then schedule the execution of a test with that firmware. During the experiment, a copy of the firmware is installed on all the motes and the testbed starts collecting information from the devices. The logging and the debugging statements are part of the firmware program and at run-time, they are sent to the testbed through a serial interface. After the experiment completion, the collected serial log messages are available for download.

All the software developed as part of this dissertation was tested on multiple testbeds. The experimental evaluations come from Indriya [35] in Singapore and FlockLab [132] in Zurich. A significant amount of experiments was conducted on Twist [75] in Berlin, Kansei [9] in Ohio, and Twonet [129] in Houston.

The remote testbeds, however, can test only some of the WSN applications. Due to their remote location, WSN applications interacting with the physical environment and using sensor measurements are difficult to evaluate and debug. For instance, sensors can fail and return invalid measurements [209]. Debugging sensors' measurements and application logic relying on this data requires an understanding of the installation environment and usually someone's presence at the testbed site or another tool proving high-quality point of reference for the physical phenomenon observed by the sensors. Similar testbed limitations exist in developing energy-harvesting WSN [79].

In Chapter 5, the limitations of the remote testbeds are addressed by a software framework that greatly simplifies the installation of a local testbed. Ultimately, this allows the designer to be present in the physical environment during the software experimentation [184].

Currently, for a research contribution to be recognized, it is necessary to either deploy a testbed when the physical environment is to be understood or controlled, or to use a remote one. The WSN research contributions often rely on either TinyOS or Contiki software. A common testing application benchmark is data collection, with authors addressing challenges with either processing and analyzing the data representing measurements from the environment [101; 22], or studying new techniques for network routing and distributed computation. One of the fundamental key metrics in the experimental evaluation is power consumption, which is reported as the percentage of time during which devices are in the active mode and the sleep mode.

2.3 Data Collection Application and The Art of Doing Nothing

The major application of the WSN is the collection of sensors measurements. With wireless technology the sensors can be deployed without wires, which simplifies and reduces the installation and maintenance costs. However, there are two major research challenges related to WSN. First, in the WSN data is transmitted over the wireless medium, characterized with a limited data rate, a significant chance of packet loss, and unstable network topology. These issues require the use of multi-hop data routing protocols that can establish paths from every mote toward the sink or a gateway to another network. Second, more than in the desktop and cloud computing [72; 92; 97], the energy saving and system duty cycling plays an imperative role in the WSN. Saving energy is crucial for economical soundness of a WSN investment, which initial costs can be returned after years of successful operation. Thus, the software running on a WSN aims at the smallest energy consumption in the idle state, and at the scalable and robust routing when application collects sensors' measurements.

Different multi-hop network routing protocols have been proposed for the WSN. CTP [62] was the first well-tested, TinyOS-default data collection protocol. On the Motelab testbed with 131 motes, CTP delivered over 97.1% of packets, with 4.6% average duty cycle. BFC [164] improved CTP's energy consumption by addressing the overhead coming from transmitting broadcast mes-

sages in duty cycling radios. Then, ORW [113] decreased duty cycle by 50% and shortened delay by 90% with respect to CTP, thanks to opportunistic routing, where motes forward data toward destination by choosing among few potential parent nodes, instead of a single one. CTP inspired IETF to define a standard for the network protocol of low-power wireless networks, called RPL [200]. ORPL [44] enhances RPL with opportunistic routing and achieves over 99% packet delivery with 0.48% duty cycle. DualMOP-RPL [158] addresses the performance challenges in the RPL downward routing, from the gateway back to the sensor motes.

Another research angle on improving data collection addresses the challenges in the lower layers of the network stack, the physical and the MAC. One factor that has a major impact on duty cycle is the interference with other radio transmissions. AEDP [170] minimizes the false positive transmission detections by dynamically adjusting the mote's CCA threshold. ZiSense [212] differentiates between ZigBee and other radio transmissions by studying time-domain RSSI sequence. ContikiMAC [40] uses two short CCA checks to skip non-802.15.4 transmissions. It is the default CSMA MAC protocol for Contiki OS. TinyOS has similar default duty cycle approach, CSMA BoXMAC protocol [151], but this one is susceptible to false-positive detections [153; 170; 212]. In duty cycling WSN, both MACs are sender-initiated protocols: a mote that wants to send a message initiates communication. Alternatively, one can use a receiver-initiated protocol such as A-MAC [45], where the mote that wants to send a message waits for the receiver to initiate communication. A-MAC improves duty cycle and saves more energy in those installations that are exposed to WiFi interference.

There are also WSN deployment techniques tailored specifically to data collection and keeping the radio turned-off for as long as possible. Dozer [20] uses local neighborhood TDMA MAC protocol to achieve low duty cycle operation. On a tree-based data-collection topology, children and parent motes agree on their transmission schedule. On a 40-mote testbed, collecting data every 2 minutes, authors report 0.2% duty cycle. Koala [153] uses a dedicated gateway to force the WSN into a deep sleep state between bulk-data downloads at the user specified rate. Authors report 0.2% duty cycle on 24 Tmote Sky testbed, requiring 30 seconds to wake up the network and start downloading data. DISSense [27] adapts the radio duty cycle between the period when data is collected from the network and the period without application activity. On a 15-node testbed, with application collecting data every 60 minutes, authors report 0.15% duty cycle. Finally, in a

light-traffic data collection scenario, LWB [54] running on Glossy [55] reports 0.41-0.48% duty cycle on the FlockLab testbed.

The experimental results of the contributions presented in this dissertation include tests with the data collection application and comprise of performance evaluations about the WSN energy efficiency. These experiments show the feasibility of this work in addressing the most important WSN applications and run-time objectives. Using data collection and duty cycling as benchmarks, the final experimental conclusions are that the software developed as part of this dissertation has low run-time overhead and it can achieve a competitive performance, while providing WSN abstractions for macro-programming.

Chapter 3

Heterogenous Applications

This chapter discusses the design-time and run-time challenges in programming a WSN to operate with multiple tasks on the same hardware installation. I present the problem of running two heterogeneous applications on the same WSN. The first application requires a low-rate reliable many-to-one data collection while the other one needs a point-to-point low-latency bulk-data streaming. Using a remote testbed, I show that these two applications cannot efficiently execute on the same WSN running on a fix network communication architecture. Instead, these applications require different types of communication services.

I introduce the building blocks of a framework that supports the execution of the two heterogeneous applications. The framework makes it possible to specify, at design-time, distinct network, MAC and radio protocols for each application as well as the events and policies triggering the context switch between the two application executions. At run-time, in response to these events and according to these policies, the WSN automatically switches between the two applications and dynamically reconfigures the network protocols to adapt to the applications' different communication requirements.

The initial results of this work were demonstrated at the ACM Conference on Embedded Networked Sensor Systems (SenSys) in 2011 [181]. The completed work was presented at the International Conference on Distributed Computing in Sensor Systems (DCOSS) in 2013 [182], and published with coauthors Omprakash Gnawali from the University of Houston and my advisor Luca P. Carloni.

3.1 Introduction

Indoor climate monitoring and control, intrusion detection, and energy-use monitoring are examples of Wireless Sensor Network (WSN) applications being deployed in large numbers. Often, each new application requires installation of a dedicated WSN. However, researchers have realized that it is infeasible to deploy a separate WSN for each application.

We propose a WSN framework that supports the execution of different applications at different times. We motivate and demonstrate our framework by presenting the combined deployment of two heterogeneous applications for indoor monitoring of a building environment on the same WSN. The deployment must satisfy the following requirements:

1. Minimize the number of WSN nodes deployed in the building.

2. During normal operation, the network must reliably collect climate data (e.g., temperature) to a single server while remaining energy efficient. We call this application *Collection*.

3. When an emergency event occurs in a particular zone of the building (e.g. a smoke sensor goes off), the network must rapidly transmit a sequence of images from this zone to the server. We call this application *Firecam*.

The first requirement (1), common to many other WSN applications, stems from physical, logistical, and cost considerations. While compressive-sensing and optimal sensor placement partly address this requirement, our approach shares nodes across applications to reduce the number of required nodes. The second (2) and third (3) requirements are specific to the *Collection* and *Firecam* applications.

In our desire to leverage prior work to meet the requirements of our target applications we focus on a few choices:

- Run different dataflow programs corresponding to *Collection* or *Firecam* at different times on top of the same network, link, and physical layer protocols, as in Tenet [157].

- Re-program the WSN using systems such as Deluge [87] when we switch from *Collection* to *Firecam*.

- Re-configure the MAC parameters with systems such as pTunes [215] to optimize performance as the traffic pattern changes between running *Collection* and *Firecam*.

Tenet and pTunes do not allow the two applications to run on their preferred protocol stack. Deluge, however, takes several minutes to reprogram the nodes, leading to unacceptable delays while transitioning from *Collection* to *Firecam*. We also find and show (later in the chapter) that the selection of different protocols or tuning the parameters of a single layer (e.g., MAC) misses the opportunity to comprehensively optimize network performance at each operational phase.

To overcome these limitations we developed Fennec Fox, a framework to dynamically reconfigure a WSN to support different applications at different times. To perform optimally, these applications depend on different network and MAC protocols. By providing a way to dynamically select and configure each component of the protocol stack, Fennec Fox allows us to leverage these existing works and support execution of heterogeneous applications on a single WSN.

Our approach consists of two steps. At design-time, for each application we can specify a distinct protocol stack (consisting of a network, a MAC, and a radio protocol) as well as the policies that govern the WSN reconfigurations and the events that trigger them. Then, at run-time, the WSN automatically reconfigures itself in response to these events and according to these policies. These two steps result in the dynamic scheduling of the execution of different applications, each supported by its own optimized network, MAC, and radio protocols.

Our main contributions include:

- Design of a language and tool-chain to configure a network protocol stack to support execution of an application and the conditions under which different applications with their corresponding protocols should execute.

- Implementation of Fennec Fox to demonstrate the feasibility of executing *Collection* and *Firecam* on top of their preferred protocol stack in a single WSN.

- Evaluation of Fennec Fox on a 119-node testbed, showing that dynamic reconfiguration is not only feasible, but also quick, efficient, and expandable to a large number of applications and various protocols.

The rest of the chapter is organized as follows: Section 3.2 presents evidence that *Collection* and *Firecam* should run on different network stacks for optimal performance. This motivated the

development of the Fennec Fox framework, whose main components are discussed in Section 3.3. In Section 3.4, we show experimental evaluation of the Fennec Fox framework. We present related work in Section 3.5.

We want to run two different sensor network applications in our office building.

Collection is the default WSN application, in the sense that is executed "continuously" during the normal network operations: it monitors the building's environment by collecting various kinds of information (e.g. temperature, humidity, light, and, possibly, also people occupancy through PIR and camera sensors) from the WSN nodes that are distributed almost uniformly across all the various zones in which is partitioned the building. The collected information can be used for various purposes, such as improving the operation of the HVAC system or saving the power consumed by the building. As part of this application, every WSN node periodically sends a message with the collected sensors' sample to a collective node hosted on a server (the *sink*.) These messages are small, just a few tens of bytes, but transmission reliability is important, i.e the sink is expected to receive them with a high delivery ratio[1]. The period of the transmission may vary depending on the size of the building and the required granularity of the measurements. Typical values of the period are between 1 and 3 minutes. When the nodes are not transmitting data, the WSN is duty-cycled to minimize power consumption.

The second application of interest, *Firecam*, is executed much more rarely as a consequence of an emergency event. Specifically, when a smoke detector or a security sensor goes off in a particular zone of the building, a node (or a very limited number of nodes) in that zone takes a series of pictures and sends this stream of pictures to a sink node, which may or may not coincide with the same sink node of the other application. Thus, differently from the previous application, in the case of *Firecam* we are interested in the transmission of large amounts of data (the size of a single picture is about 76k bytes) on a point-to-point connection between two nodes that are typically located in two distant zones of the building. One of the purposes of *Firecam* is to assist emergency/security personnel to immediately assess the gravity of the potential problem[2].

---

[1]The delivery ratio is defined as the ratio of total number of received messages over total number of sent messages.

[2] This is indeed a practical issue since many emergency alarms are often the results of false positive sensor readings. It is thus helpful to have a mechanism that can quickly confirm the occurrence of real problems through the real-time delivery of pictures of the particular zones.

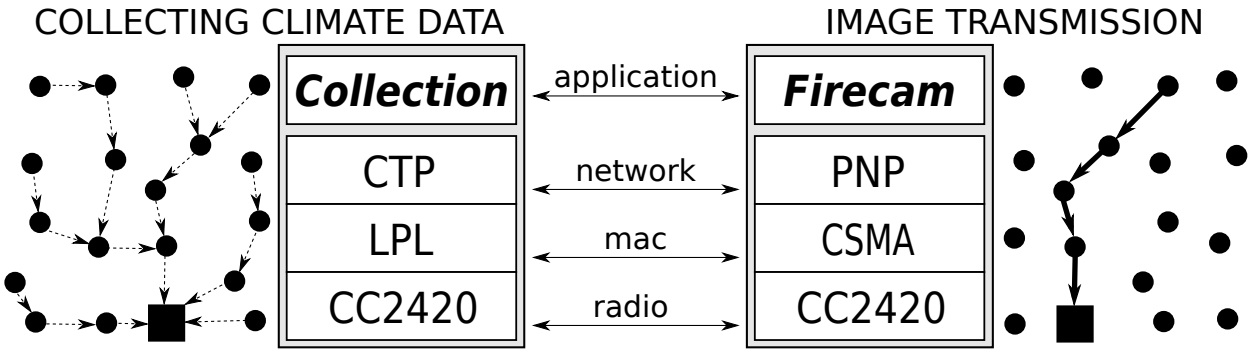


Figure 3.1: Two different applications and their corresponding protocol stacks. In the sequel we will use the term *CTP stack* and *PNP stack* to denote the protocol stacks required by *Collection* and *Firecam*, respectively.

3.2 One Network, Two Applications

Figure 3.1 illustrates the main characteristics of the two applications in terms of communication patterns across the WSN. It also shows the choice of the protocols that are optimized to execute each of them. In particular, the state-of-the-art data-collection protocol CTP [62] is best suited to support the *Collection* application because it achieves a delivery ratio close to 100% while being also very power efficient. It does so by establishing a network-tree topology and routing the packets with the applications' small messages from the various nodes toward the sink. CTP is a multi-hop network protocol that relies on the services provided by the MAC and radio protocols, which focus on a single transmission between two nodes. CTP was designed to run on top of a CSMA MAC protocol, which attempts to avoid transmission collisions by sensing presence of other radio communications and introducing random transmission delays. Also, the CSMA MAC protocol is typically augmented with a Low Power Listening (LPL) mechanism to duty-cycle the WSN. The quality of a single-hop transmission is further supported by radio services that provide clear channel assessment (CCA), auto acknowledgement, and automatic CRC error-detection.

In the case of *Firecam*, instead, the efficient transmission of a stream of pictures from one particular node to the sink requires to quickly establish a multi-hop path between them. Assuming that a picture size is 240×320 pixels and that each pixel is encoded as a single byte, the transmission of a single uncompressed picture requires the transfer of 76800 bytes. We can partition such picture in 768 packets each storing 100 bytes of picture data and a 4-byte sequence number that is

necessary to allow the picture reconstruction at the sink. The *Parasite Network Protocol* (PNP) is a network protocol that can efficiently support the *Firecam* application by forwarding the packets at a constant rate over a fixed path. Similarly to the protocols proposed by Kim et al. [100] and Österlind et al. [155], PNP relies on the presence of another protocol that establishes the multi-hop path and, in order to achieve a high-throughput, assumes the absence of other network traffic. Also, PNP works more efficiently on top of a simplified MAC protocol, where most of the CSMA functionality is disabled, without CCA and CRC checks, and with a radio protocol where the auto acknowledgement is also disabled.

Next, we discuss experimental results that confirms the following important fact about the protocol stacks shown in Figure 3.1: *each of them supports well the corresponding application, for which it has been optimized, while supporting poorly the other application.*

Experimental Setup. The School of Computing building at the National University of Singapore is a three-floor building that has been instrumented with a WSN testbed called Indriya [35], which consists of 119 active TelosB motes [163]. TelosB has a CC2420 radio, 8 MHz CPU, 10 KB RAM, 48 KB of program memory, and is a widely used hardware platform in WSN research. In the first set of experiments, which are discussed in this section, we remotely programmed the Indriya motes to support the *Firecam* and *Collection* applications separately without WSN reconfiguration (the reconfiguration experiments with Fennec Fox are discussed in Section 3.4.) In all our experiments we assumed that the sink node is located at the corner of the first floor. For *Collection*, all remaining 118 nodes send data to the sink, while in the case of *Firecam*, the picture is streamed from a node located at the opposite corner of the building, on the third floor. The path between two opposite corners requires 7 to 9 hops. All experiments have been completed multiple times, over a period of two-weeks, during day and night hours, in midweek and weekend days.

PNP Works Better than CTP to Support *Firecam*. Figure 3.2 shows how fast a picture from *Firecam* application can be streamed over a WSN with the PNP stack discussed above. In particular, it reports the results of multiple experiments to show how the network throughput (measured as the number of packets received at the sink per unit of time) and the delivery ratio (as previously defined) vary as function of the inter-packet transmission interval, which is varied at the step of 5ms in the range [5ms, 50ms]. For inter-packet transmission intervals equal or more than 30ms, the packets arrive with delivery ratio close to 100%. While the delivery ratio is still

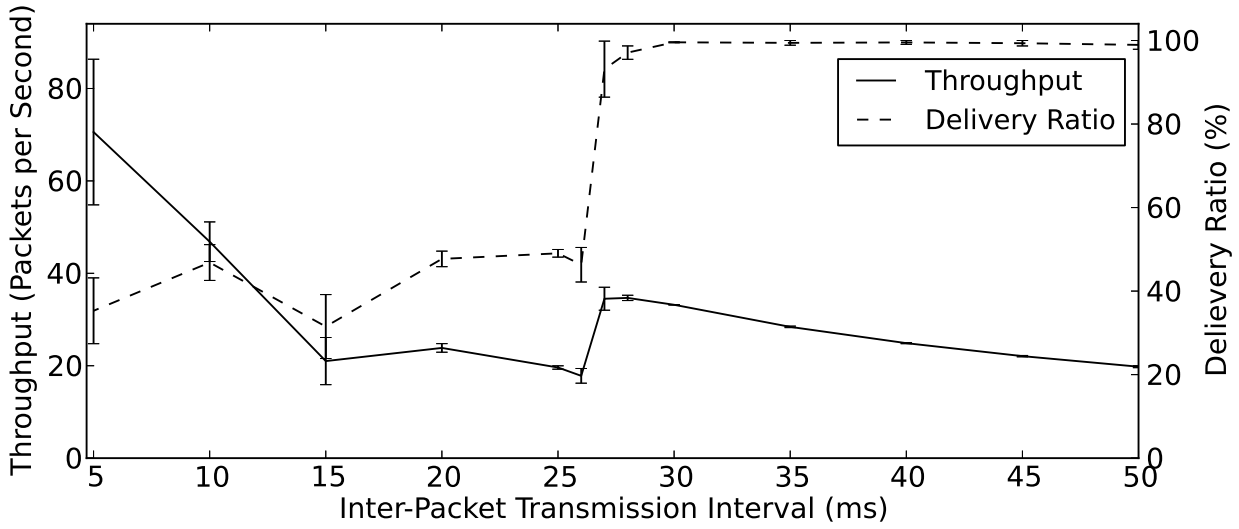


Figure 3.2: Throughput and Delivery Ratio during operation of the PNP stack.

97.4% for an interval value equal to 28ms, it drops to 50%, due to transmission collisions, for a 26ms interval value. The network throughput values are 33.96, 35.63, and 35.32 packets per second for 30, 28, and 27ms interval values, respectively. In summary, we consider a 28ms inter-packet transmission interval as the most adequate to transmit a picture as it allows a successful transfer within 21.5 seconds.

Next, we study how fast a picture from the *Firecam* application can be streamed over a network running the CTP stack with a CSMA MAC using 10 jiffies random backoff, CCA, CRC, and radio's auto acknowledgment. Notice, that we purposely disabled the LPL mechanism because it does not provide any help for the type of transmission that characterizes the *Firecam* application. Figure 3.3 shows the corresponding experimental results in terms of network throughput and delivery ratio as the inter-packet transmission interval is varied at the step of 10ms in the range [10ms, 100ms]. It is clear that with this WSN configuration the *Firecam* application suffers a low delivery ratio. While streaming a packet every 100ms yields a delivery ratio close to 100%, this drops considerably to 88% and 60% for lower inter-packet transmission intervals equal to 50ms and 30ms, respectively. As highlighted in Figure 3.3, there is a clear throughput gap between the performance of the two protocol stacks of Figure 3.1 when running the same *Firecam* application. The top line marks the best throughput achieved by the PNP stack, which delivers over 97% of packets with a throughput of 36.52 packets per second. The CTP stack, instead, can only achieve 88% delivery ratio at the

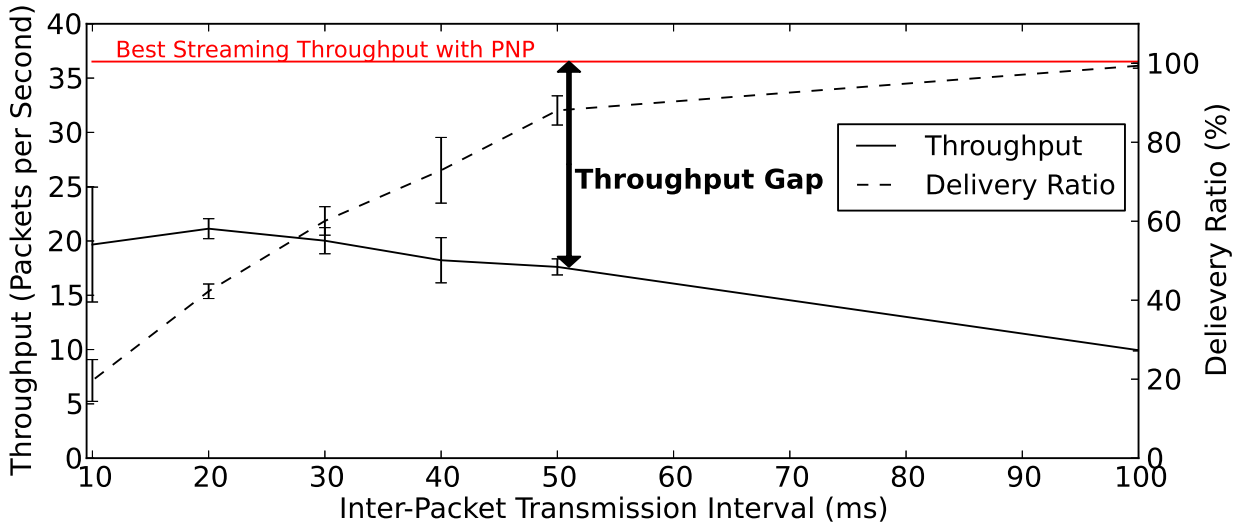


Figure 3.3: CTP with CSMA does not support high point-to-point throughput.

50ms inter-packet interval, with a throughput of 17.61 packets per second: at this rate, a picture is transmitted in 38.4 seconds, which is more than twice as long as taken when streaming it with the PNP protocol.

In summary, based on the results of Figure 3.2 and Figure 3.3, we conclude that the *Firecam* application clearly benefits from a WSN deployment that uses the PNP stack. Next, it is natural to study how well this protocol stack can support the very different *Collection* application.

CTP Works Better than PNP to Support *Collection*. Figure 3.4 compares the packet delivery ratio for the messages of the *Collection* application for three different configurations of the WSN protocol stack: CTP with CSMA and radio support, CTP with CSMA and Low Power Listening (LPL) duty-cycling at 100ms, and the PNP stack discussed above (i.e. without CSMA, CCA, CRC and acknowledgements.) Data are reported for three different transmission rates: 3 minutes, 1 minute, and 30 seconds. As expected, CTP achieves close to 100% delivery ratio. Even when LPL is enabled, CTP still performs well unless the sending rate becomes too high (the delivery ratio drops to 60% only when the 119 nodes are sending sensor measurements every 30 seconds.) The network configuration with PNP, instead, struggles to successfully deliver messages, as more than 70% of packets are lost. We conclude that traditional WSN applications for collecting sensor information cannot be effectively supported by the PNP stack.

The Need for Dynamic Reconfiguration. The above empirical study shows that two ap-

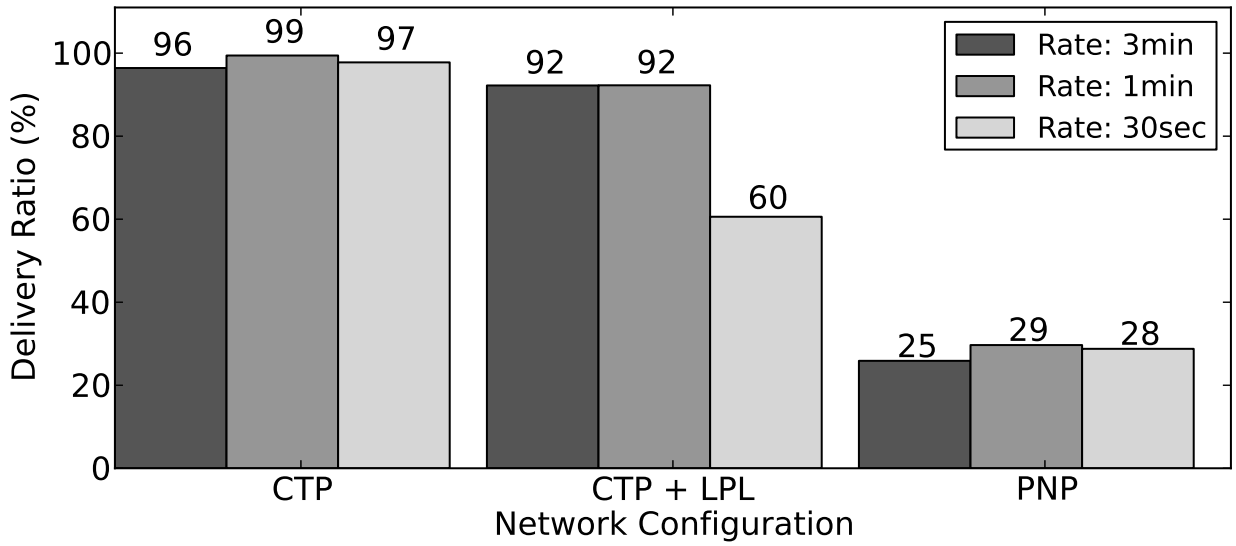


Figure 3.4: PNP cannot support the same many-to-one delivery ratio as CTP.

plications which have very different traffic characteristics require two different protocol-stack configurations in order to be properly supported. While all the experiments discussed so far have been run separately, we are interested in understanding to which extent the same WSN can effectively support two different applications such as *Firecam* and *Collection*. Running such heterogeneous applications with different network communication requirements is difficult because there is no WSN system that allows switching MAC protocols at runtime[3]. *Notice that these two applications cannot run simultaneously in an effective way. First, a WSN cannot run simultaneously the same MAC with different configurations. Second, a network protocol like PNP assumes there is no other network traffic.*

On the other hand, we are focusing on a heterogeneous application scenario that does not require simultaneous execution of the two applications. Instead, we are interested in a WSN that can run *Collection* as the default application and switch to running *Firecam*, which has a higher priority, only when an emergency event occurs. In other words, we want to deploy a WSN that: (i) can support multiple applications at different times and (ii) at any given time it uses the protocol stack configured to run those network, MAC, and radio protocols that are optimized for

[3]pTunes [215] only allows to reconfigure MAC parameters and Deluge [87] can change the MAC by reprogramming the whole sensor node firmware.

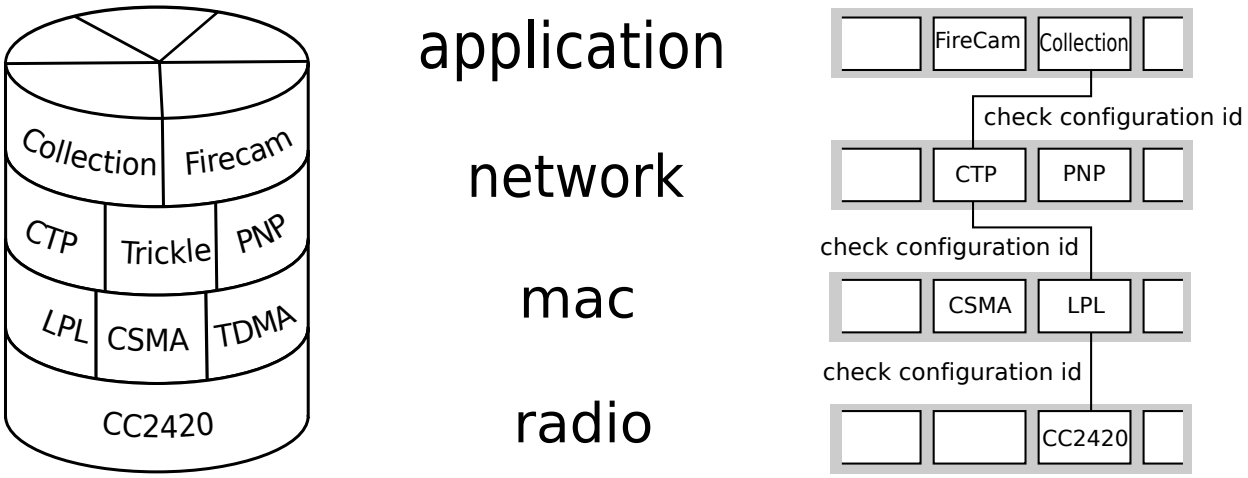


Figure 3.5: The Fennec Fox four-layer protocol stack.

the current application. Since these protocols are different for different applications, the WSN needs to dynamically reconfigure the protocol stack to support their execution. For the case of our building environment application, the *Collection* application runs on top of the CTP stack, but when an emergency event occurs, the network reconfigures to the PNP stack to support the *Firecam* application. When the emergency is over, the network reconfigures back to run *Collection*.

3.3 The Fennec Fox Framework

To support the dynamic reconfiguration of WSNs we developed the Fennec Fox framework that consists of a runtime infrastructure built around a layered protocol stack and a programming language to specify the various WSN configurations and the policy to switch among them.

Framework Definitions. Figure 3.5 shows the four layers of the stack: radio, MAC, network, and application. Each layer provides a set of *services* that are used by the layer immediately above. Each layer contains one or more modules. A *module* is a software program that provides an implementation of the services of its layer. This implementation is typically optimized with respect to some metric, such as power consumption, reliability, throughput, network routing topology, etc. Hence, depending on the particular layer, a module can be: (1) an application such as *Firecam* or *Collection*; (2) a network protocol such as CTP or PNP; (3) a MAC protocol such as CSMA or TDMA; and (4) a driver of a particular radio. A module accepts zero or more parameters, whose

values have impact on the module's execution. A *module instance* is a module with a specified set of values for its parameters. Two module instances are equivalent when they are both instantiated with the same parameter values.

A *protocol stack configuration*, or simply *configuration*, is a set of four module instances executing on the four-layer stack, one module for each layer. Each network stack configuration of a given WSN has a static, globally unique *configuration identifier* (id) defined at the WSN design-time. Two configurations are equivalent when their module instances are equivalent.

A *network reconfiguration* is the process during which the WSN switches its execution between two non-equivalent configurations, i.e., two different stacks. A node starts this process either in response to a reconfiguration request from another node or by itself as a result of an internal event, sensor readings, or an an occurrence of a periodic event.[4] Once initiated, the nodes continue with reconfiguration by requesting surrounding nodes to reconfigure as well. During reconfiguration, a node stops all the modules running across the layers of the stack and starts execution of the modules defined in the new configuration.

Framework Implementation. The Fennec Fox software running on each node is implemented in nesC [61] on top of the TinyOS operating system [127]. The software stores information about various protocol stack configurations, events triggering network reconfiguration, statically linked layers' modules and information about parameters' values that are passed to each module when it starts execution. Each module has to comply with the Fennec Fox standardized interfaces, i.e. a module must have a management interface allowing the framework to start and stop execution of the module and it must comply with the interfaces of its layer.

The network protocol stack is implemented as a set of switch statements, which direct function calls and transfer packets among the modules based on the configuration id, as shown in Figure 3.5. The id determines every function call made outside of the module's layer. To allow a radio driver to dispatch packets to the appropriate MACs, each radio defines location in a packet where it stores the configuration id, e.g. CC2420 radio driver stores the id's value in the Personal Area Network field of the IEEE 802.15.4 header [90]. The value of the packet's id is set to the id of the configuration of the stack in which that packet was created.

[4] In this chapter, we do not focus on how the nodes decide to initiate reconfiguration. Fennec Fox provides mechanism to reconfigure the stack once such a decision is made by a node or a group of nodes.

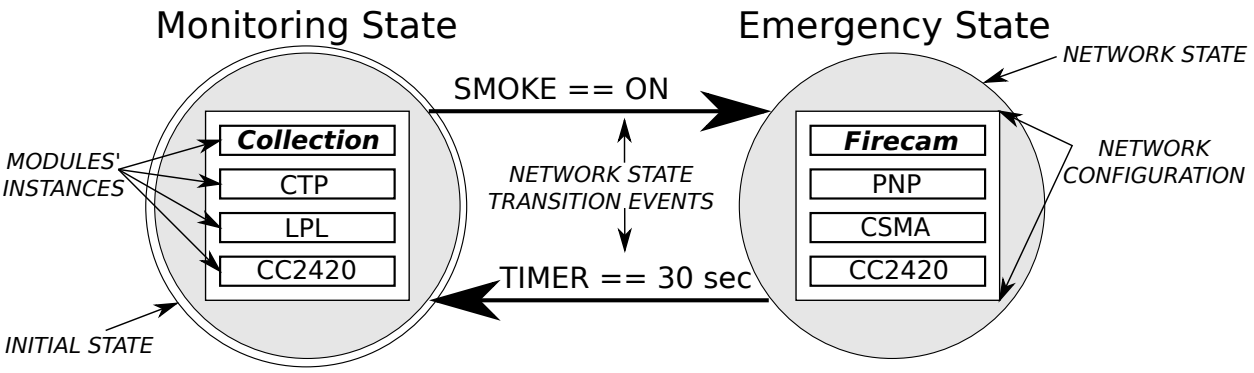


Figure 3.6: A FSM model of a WSN supporting the *Collection* and *Firecam*.

Modeling Reconfigurations with FSMs. The evolution of the behavior of a WSN that can dynamically reconfigure itself through the Fennec Fox framework can be captured in a simple way by using the Finite State Machine (FSM) model of computation. In particular, each protocol stack configuration can be modeled with a distinct state of the FSM and the process of reconfiguring the WSN between two particular configurations can be modeled with a transition between the corresponding states.

For example, Figure 3.6 shows the FSM that models the reconfiguration of a WSN supporting the two applications as discussed in Section 3.2 with the optimized stacks shown in Figure 3.1. The FSM has two states. The *Monitoring* state, which is also the initialization state, models the execution of the *Collection* application on top of the CTP stack, with the MAC and radio configured to minimize power dissipation and to avoid packet collisions. The *Emergency* state models the execution of the *Firecam* application on top of the PNP stack, with the MAC and radio configuration aimed at minimizing transmission delay and maximizing throughput. Further, the state transitions model the conditions that govern the reconfiguration of the WSN. The transition from *Monitoring* to *Emergency* specifies that this reconfiguration must occur when the smoke detector of a WSN node goes off so that the *Firecam* can start streaming a picture from the corresponding zone in the building. The transition from *Emergency* to *Monitoring* specifies that the opposite reconfiguration must occur when a certain time period has passed since the network has switched to the *Emergency* state: in this example, after a period of 30 seconds the network is brought back to execute the *Collection* application.

High-Level Programming of WSN Reconfigurations. To simplify the deployment of

reconfigurable WSNs, we developed Swift Fox, a new domain-specific high-level programming language that has its formal foundation on the simple FSM model described above. Using Swift Fox, it is possible to specify at design-time the behavior of a self-reconfiguring WSN, by scheduling the execution of each application and indicating the corresponding supporting stack configurations. A Swift Fox program allows us to control the four stack layers for each configuration by instantiating modules, initializing module parameters, and assigning unique ids to each configuration. Further, for each configuration we declare a *configuration priority level*, which plays an important role when multiple distinct reconfigurations occur at the same time in the network, as discussed below.

The semantics of the Swift Fox language supports the declaration of the sources of reconfiguration events and the threshold values that must be matched for an event to fire. The source of an event may come from a timer or a sensor. Boolean predicates can be specified using the basic relational operators (e.g. $==$, $<$, $>$) to compare sensor measurements and timer values with particular threshold values. The event-condition predicates are compiled into code that at runtime periodically evaluates the expression value. When the value is *true*, the occurrence of the event is signaled. The network FSM model is programmed by combining network state declarations with the event-conditions to form policy statements. Each policy statement specifies two network configurations and an event triggering network reconfiguration from one configuration to another. A Swift Fox program is concluded with a statement that specifies the initial configuration of the WSN.

The Fennec Fox software infrastructure relies on the definitions of the network configurations written in the Swift Fox program. This includes not only the logic to capture possible reconfigurations but also the list of modules that are executed across the layers of the stack for each particular configuration, i.e. which application and which network, MAC, and radio modules together with the values of their parameters. The Swift Fox programs are compiled into nesC code that links together all the modules that are specified for a given configuration and generates switch statements that direct function calls and signals among the modules.

As an example, Figure 4.5 shows a Swift Fox program for a WSN that reconfigures between the *Monitoring* and *Emergency* states according to the state transition diagram of the FSM of Figure 3.6 in order to support the execution of the *Collection* and *Firecam* applications, respectively. Lines 3-6 declare the two network configurations with ids *Monitoring* and *Emergency*. The *Monitoring*

```
 1 # Definition of network configurations
 2 # configuration <conf_d> [priority level] {<app> <net> <mac> <radio>}
 3 configuration Monitoring {collection(2000, 300, 1024, NODE, 107)
 4                           ctp(107) lpl(100, 100, 10, 10) cc2420(1, 1, 1)}
 5 configuration Emergency L3 {firecam(1000, 28) parasite()
 6                                                csma(0, 0) cc2420(0, 0, 0)}
 7 # Events: event-condition <event_id> {<source> <condition> [scale]}
 8 event-condition fire {smoke = YES}
 9 event-condition check_if_safe {timer = 30 sec}
10
11 # Policies: from <conf_id> to <conf_id> when <event_id>
12 from Monitoring goto Emergency when fire
13 from Emergency goto Monitoring when check_if_safe
14
15 # Definition of the initial state: start <conf_id>
16 start Monitoring
```

Figure 3.7: Swift Fox program reconfiguring WSN between two applications.

configuration consists of the *Collection* application module that starts sensing after *2000ms* since the moment it receives the start command on the management interface. From every *NODE*, the module sends messages with the sensors' measurements at the rate of *300* seconds (*1024ms*). The messages are sent to a sink node whose address is *107*. Indeed, the configuration uses the *ctp* module, which runs the CTP network protocol with a root node at the address *107*. The *Collection* configuration runs also a MAC protocol with Low-Power Listening (*lpl*), a *100ms* wakeup period and stay-awake interval, together with *10jiffies* random backoff, and *10jiffies* minimum backoff CSMA's parameters. The configuration is supported by a radio driver enabling all three services: auto-acknowledgements, CCA and CRC. The specification of the *Emergency* configuration is similar but it is characterized by a higher priority level (set to 3 while the default is 1) and by the use of PNP with all MAC and radio services disabled. Lines 8-9 declare two reconfiguration events. The first event, *fire*, occurs when a sensor detects the presence of smoke. The second event, *check_if_safe* takes place *30 seconds* after it is initiated. Lines 12-13 declare the network state reconfiguration policies: the network reconfigures from *Monitoring* to *Emergency* when *fire* occurs; similarly, it

reconfigures from *Emergency* back to *Monitoring* when the *check_if_safe* occurs. Line 16 sets *Monitoring* to be the initial state.

The same Swift Fox program is deployed on every node of the given WSN. While Swift Fox allows us to program a reconfigurable WSN, the language does not allow programmers to specify how a particular node detects an event, how it reconfigures itself, and how it can trigger the reconfiguration of all the other nodes in the network. Indeed, Swift Fox is meant to provide a high-level abstraction that intentionally hides the underlying mechanisms governing the WSN reconfiguration.

Runtime Network Reconfiguration. A node decides to reconfigure when the result of an event matches the reconfiguration policy in the Swift Fox program. Then, the node requests other nodes to reconfigure by broadcasting a *Control Messages* (CM), a single 4-byte packets that contains the id and the *sequence number* of the new configuration. The sequence number is incremented by one after each network reconfiguration. Based on the sequence number, nodes can distinguish a new configuration from an old one. As a result of the nodes re-broadcasting CM packets to other nodes during reconfiguration, the whole WSN reconfigures itself.

The network reconfiguration process requires dissemination of CM packets in the presence of various MAC protocols scheduled to run on the stack at a given time. CM packets are distinguished from other packets by their own configuration id, which allows radio drivers to dispatch CM packets to Fennec Fox. To enable transmission of CM packets during operation of other MACs or radio duty-cycling, Fennec Fox monitors the radio status together with function calls and packets crossing the layers of the stack, making decision on when CMs should be transmitted such that other nodes will receive the message, i.e. the CM broadcasts are suspended when a radio is turned off or other transmissions are ongoing.

The CM dissemination process has been successfully tested to reconfigure the network among TDMA, CSMA, and duty-cycling versions of these protocols However, TDMA to TDMA reconfigurations may not be successful when both MACs duty-cycle with the same period but end up at different offset. This problem is mitigated by introducing a transition configuration with a CSMA MAC that runs between the two TDMA-based configurations.

The CM broadcast functionality, which co-existing with other MACs, supports network reconfiguration operating on a modified Trickle [128] algorithm. First, no messages are disseminated when network reconfiguration does not take place and all nodes in the network run the same con-

Algorithm 1 Broadcast Control Process (BCP)

```
1: retry ← r
2: while retry > 0 do
3:    counter ← 0
4:    WAIT(d)
5:    if  counter < t then
6:       BROADCAST_CM
7:    end if
8:    retry ← retry − 1;
9: end while
```

figuration. Second, to ensure that all nodes run the same stack after switching their state, not only the sequence number but also the content of the CM is used during network reconfiguration.

To distribute CMs a node follows the *Broadcast Control Process* (BCP), which is specified as Algorithm 1. In particular, the node attempts to broadcast the CM every d ms (line 4,6). A node abstains from broadcasting when it receives t identical CMs sent by other nodes within the last d ms (lines 5-7)[5]. The BCP terminates after r broadcasts attempts (lines 1-2, 8-9).

A node enters the BCP as a result of one of three possible situations. First, after a node has completed a stack reconfiguration it enters BCP to request other nodes to switch to the same configuration. Second, when a node receives a data packet with a configuration id that is different from its current configuration, it assumes that it either missed the last network reconfiguration or the node transmitting the packet has missed it; to resolve this situation the node enters BCP[6]. Third, the reception of a CM may lead also to the execution of BCP, depending on the values of the configuration id of the new state and sequence number in the control message as well as the corresponding current values stored in the node; all these values are processed by the node executing Algorithm 2.

Algorithm 2 specifies the decision process followed by a node after receiving a new CM. First, this message is validated by checking its CRC code and the value of the configuration id that it

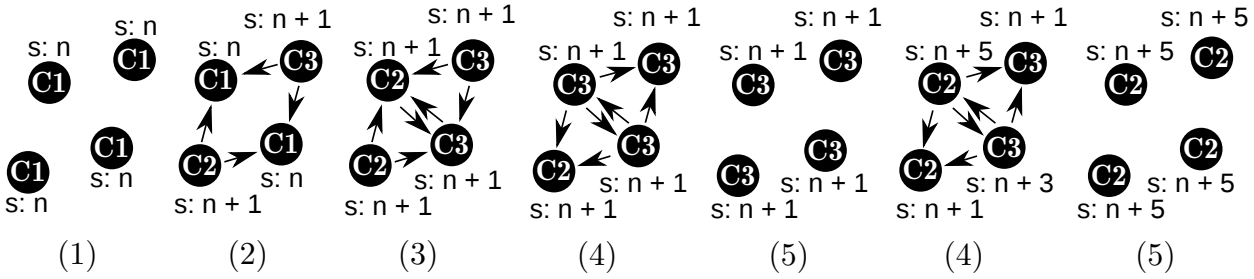
[5]This transmission suppression avoids unnecessary radio broadcasts, in a way similar to the Trickle protocol [128].

[6] Recall that every packet carries its configuration id and therefore every packet can be used to detect network configuration inconsistency.

Algorithm 2 Processing Received Control Message

```
1:  Input: msg
2:  if !CRC(msg) || msg.state ∉ ALL_STATES then
3:     EXIT
4:  end if
5:  msg_version ← concat( msg.sequence, PRIORITY(msg.state) )
6:  node_version ← concat( node.sequence, PRIORITY(node.state) )
7:  if msg_version < node_version then
8:     BCP ; EXIT
9:  end if
10: if msg_version > node_version then
11:    RECONFIGURE ; EXIT
12: end if
13: if msg.state = node.state then
14:    counter++
15: else
16:    node.sequence += RANDOM
17:    BCP
18: end if
```

carries. If either CRC fails or the configuration id value is different from all known configuration ids, which are specified at design-time, then CM is ignored (lines 2-4). The algorithm decides to run BCP or trigger node reconfiguration by comparing CM's configuration version with node's configuration version, which are computed by concatenating the sequence number and priority of the configuration id from the CM and node, respectively (lines 5-6). The comparison of both CM and node configuration versions leads to the following decisions. If CM has a sequence number less than the node's sequence number or the sequence numbers are equal but the CM's configuration has lower priority than the node's configuration, then the node enters BCP (lines 7-9). If CM has a sequence number higher than the node's sequence number, or the sequence numbers are equal but the CM's configuration has higher priority than the node's configuration, then the node switches to the new configuration (lines 10-12). If a node has the same sequence number and configuration



(a) Two events (b) Definite (c) Random

Figure 3.8: Network synchronization: (b) deterministic and (c) non-deterministic.

id as CM has, then the node increases *counter* by 1 (line 14).

When CM and a node have equal sequence numbers but different configuration ids then the network is unsynchronized. This situation occurs when two nodes simultaneously decide to run different configurations. Then these nodes start BCP with the same sequence number, but different configuration ids. This is illustrated in Figure 3.8(a) showing two nodes in the corners of the network reconfiguring from a configuration with id C_1 and sequence number n to two different configurations C_2 and C_3, both with sequence $n+1$. The nodes in the middle of the network detect reconfiguration inconsistency. If the configuration ids of the conflicting CMs have different priorities, then the network is deterministically synchronized to the configuration with the higher priority (lines 7-12). This is shown in Figure 3.8(b) where the conflict between C_2 and C_3 is solved by synchronizing to C_3 because Priority(C_3) > Priority(C_2). When the network is unsynchronized among states with undefined[7] or equal priorities then the nodes that detect the conflict increase their sequence numbers by a random value and start BCP while keeping their current configuration (lines 16-17). As shown on Figure 3.8(c) where Priority(C_2) = Priority(C_3), after randomly increasing sequence number (+5 and +3, for C_2 and C_3 respectively), the node that broadcasts CMs with the highest sequence ($5 > 3$) will synchronize the rest of the network to its own configuration, i.e. C_2.

In summary, the Fennec Fox network reconfiguration mechanism has the following properties: (1) it controls the execution of the four-layer stack and applies the specification of the network behavior given in the Swift Fox program; (2) it has zero overhead when no reconfiguration takes

[7]Recall that the Swift Fox language allows, but not mandates, programmers to specify the network configuration's priority level. By default each configuration priority level is set to the lowest possible value.

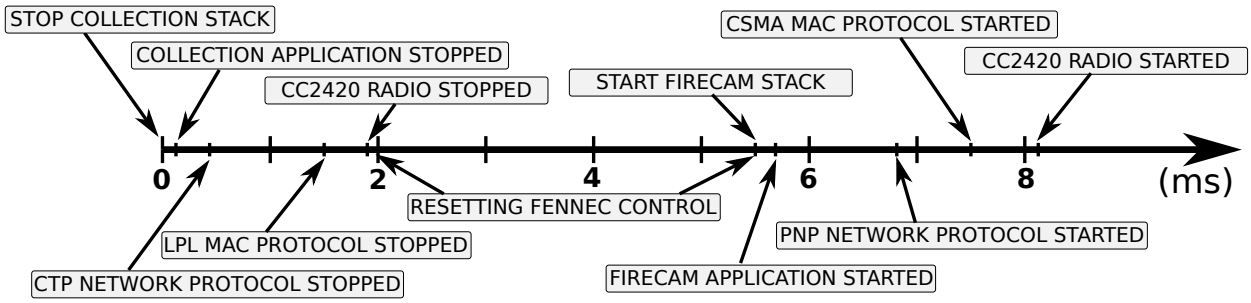


Figure 3.9: Protocol stack reconfiguration from *Collection* to *Firecam*.

place; (3) the network reconfiguration does not require any hardware support; and (4) it is guaranteed to resolve any possible reconfiguration conflict that may arise given the distributed nature of the mechanism.

3.4 Evaluation

The goal of our experiments is to study the feasibility and performance of dynamic WSN reconfiguration. We show the memory overhead and time that it takes to reconfigure the protocol stack on a single node. We present network reconfiguration experiments with the setup as described in Section 3.2. We demonstrate the feasibility of our approach and measure the overhead of reconfiguration between stacks running different MACs. We also study BCP algorithm configurations that successfully disseminate CM packets.

Code and Memory Overhead. On TelosB, the reconfiguration protocols and mechanisms introduce an overhead of 4.7 KB of ROM and 0.2 KB of RAM. The Swift Fox program with both *Collection* and *Firecam* requires 28.9 KB of ROM and 5.6 KB of RAM.

Single-node Reconfiguration Delay. Figure 3.9 shows all the events and their timings when a node switches from running *Collection* to running *Firecam*. Once reconfiguration is initiated, Fennec Fox first stops the currently running *Collection* application module and then stops the CTP network protocol, LPL MAC, and CC2420 radio modules. This process requires a total of 1.969ms. Next, the reconfiguration engine is reset, which takes 3.469ms, of which 2.9375ms is spent resetting the radio device. Finally, Fennec Fox starts the CC2420 radio, CSMA MAC, PNP network protocol, and *Firecam* application, which takes a total of 2.686ms. The whole network

stack reconfiguration takes 8.125ms. We observe similar reconfiguration delays among other WSN configurations.

WSN Switching between *Collection* and *Firecam*. We first determine if it is feasible to reconfigure a network between *Collection* and *Firecam* applications correctly, quickly, and efficiently using the proposed Fennec Fox framework. In these experiments, the BCP algorithm (Algorithm 1) runs with $d = 18$, $t = 2$ and $r = 1$. We set node 107 located at one corner of the testbed building to be the sink node.

Feasibility of Reconfiguration. First, we run repeatedly (36 times) the following experiment: after *Collection* has executed for 5 minutes, a random node on the network triggers reconfiguration to *Firecam*. The results show that on average 99.98% of the nodes complete reconfiguration by successfully switching from *Collection* to *Firecam*. This demonstrates the feasibility of our approach. In fact, as discussed below, even in a duty-cycled network, 99.5% of the nodes are successfully reconfigured.

Multiple Reconfigurations. The next question is whether our system is robust enough to perform multiple reconfigurations. We performed the following sequence of tasks for 100 minutes: run *Collection* for 15 minutes before letting a node trigger reconfiguration to *Firecam*; then, after 1 minute, the network is reconfigured back to run *Collection* and the process is repeated. The lower graph in Figure 3.10 shows the percentage of nodes executing *Firecam* at a given time: except during the transition between the configurations, all the nodes are running either *Collection* or *Firecam*. This transition occurs 12 times as shown in Figure 3.10. As the network transitions between execution of *Collection* and *Firecam*, we expect to see the network throughput observed from the sink to transition between low and high throughput. This is confirmed by the results shown in the upper graph of Figure 3.10. reporting the throughput sampled every minute. The timing of these transitions match the timing of reconfigurations. This provides additional evidence that the reconfigurations indeed make the network transition between two completely different applications; furthermore, the applications and their protocol stack are not impaired by the proposed reconfiguration mechanism.

Network-wide Reconfiguration Delay. The graphs that are embedded in the lower graph of Figure 3.10 show the network reconfiguration at time scale of milliseconds, thereby highlighting how much time it takes for the network to transition between two configurations. The first embedded graph shows the number of nodes executing *Collection* just before the reconfiguration: a rapid

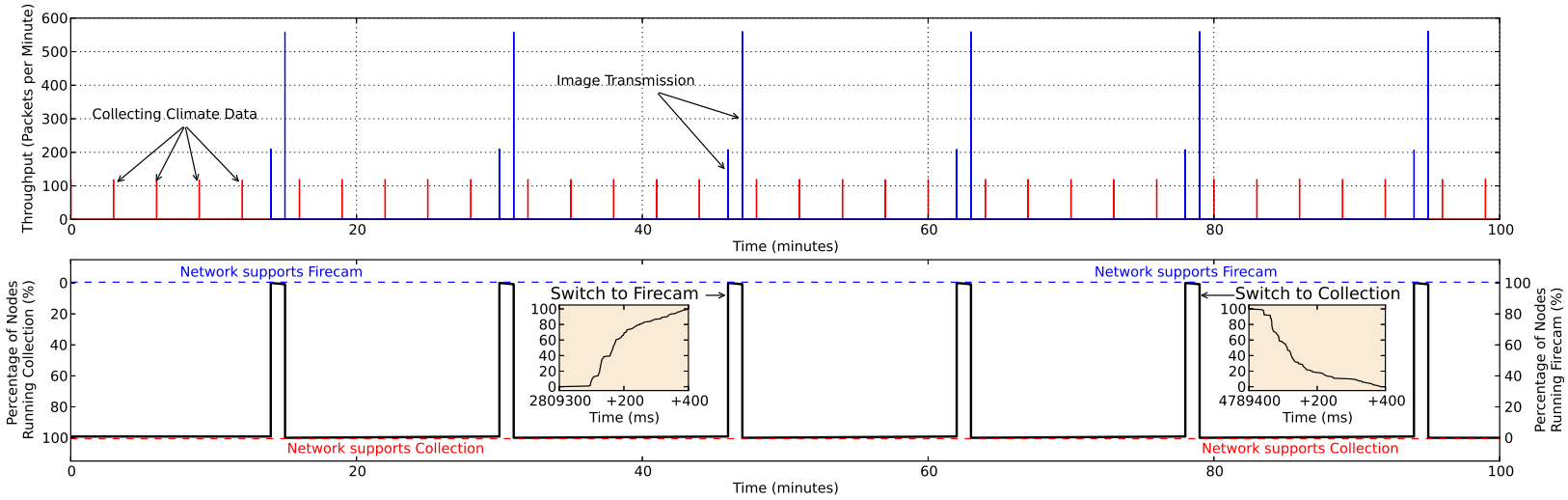
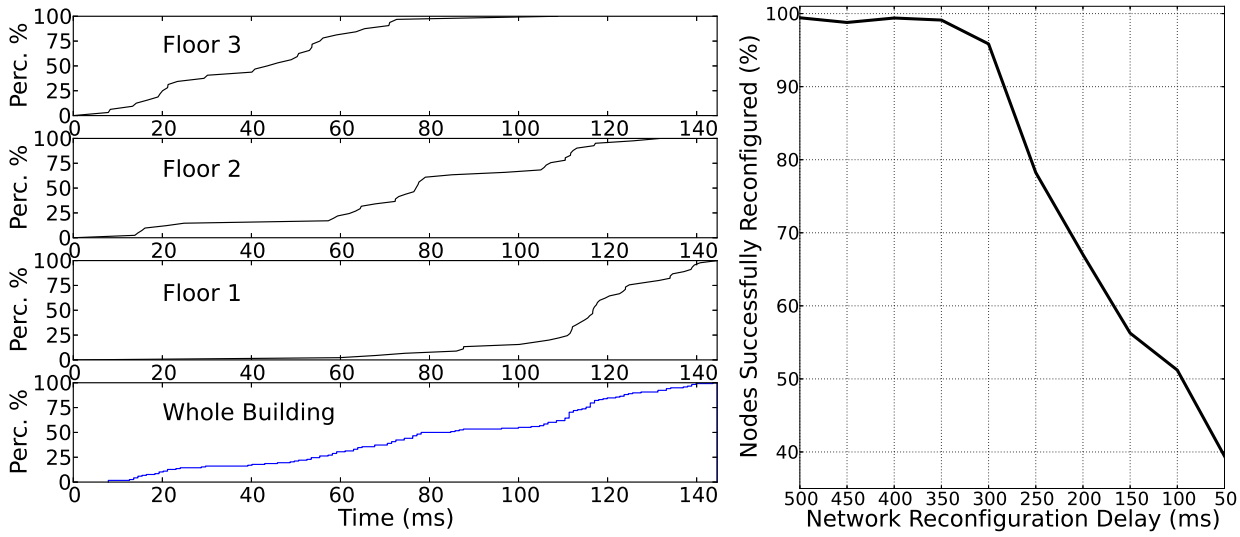


Figure 3.10: A 100 minute run of a network reconfiguring between the *Collection* and *Firecam* applications. The red/low bars, each with 119 packets, correspond to moments when every node reports sensor measurements. The blue/high bars, with 209 and 559 packets each pair of bars, represent situations when one node streams 768 bytes of data.



(a) Building Reconfiguration

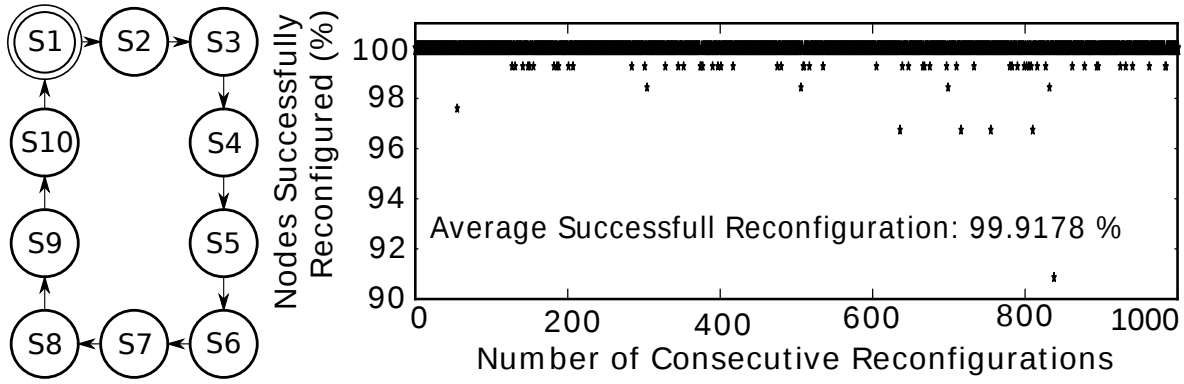
(b) Transition Rate

Figure 3.11: Reconfiguration performance with radio duty-cycled.

reconfiguration of 80% of nodes occurs within less than 100ms, while the rest of the nodes transition in the next 200ms to start *Firecam*. Similarly, the second embedded graph shows that close to 80% of the nodes reconfigure quickly, while the remaining 20% of transitions happen with the next 200ms, to switch from *Firecam* back to *Collection*.

The reconfiguration delay depends on the network distance between the nodes being reconfigured. Figure 3.11(a) shows average results from 50 experiments where a node, located in a corner of the 3rd floor, initiates a reconfiguration every minute. For each floor, reconfigurations occur in bursts: this is due to a single broadcast packet initiating the process and being able to trigger reconfiguration on all the nodes that receive that broadcast. After running multiple experiments with nodes initiating reconfiguration placed all over the building, we found that those located on the same floor as the node that starts the process switch to the new configuration within 49.81-71.54ms; instead, the nodes in the adjacent floor take a time within 95.67-153.67ms, and the nodes that are two floors away need 134.41-141.01ms.

Maximum Reconfiguration Rate. Figure 3.11(b) shows the percentage of nodes that successfully reconfigure as function of the network reconfiguration delay, varies between 50ms and 500ms. With the reconfiguration delay not less than 350ms, almost 100% of the nodes reconfigure



(a) FSM

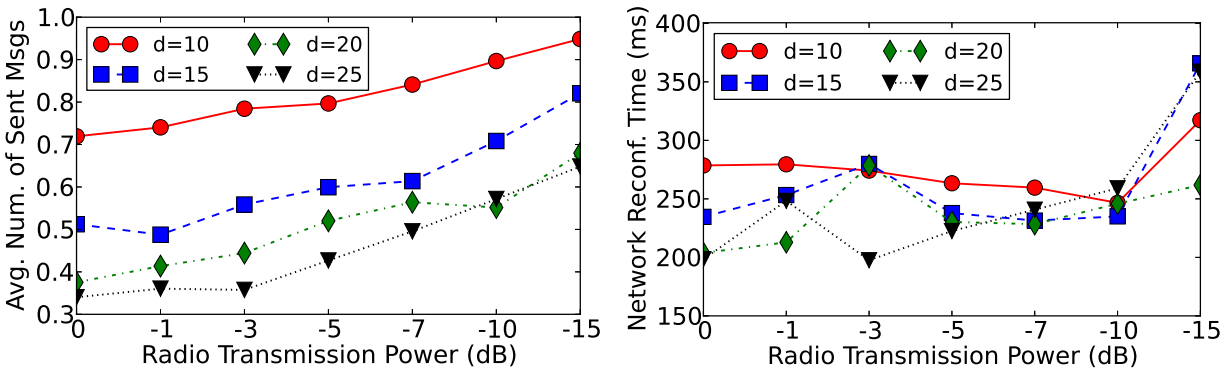
(b) 1000 Reconfigurations

Figure 3.12: Network reconfiguration firing every 500ms.

on time. However, as the reconfiguration delay decreases further, the percentage of nodes that successfully reconfigure also decreases. These results demonstrate that it is feasible to reconfigure a network successfully, if necessary, multiple times, and quickly.

Beyond Two Applications. We wrote a Swift Fox program with 10 configurations of simple functionality to emulate 10 different applications. Figure 3.12(a) shows the model of the network with a reconfiguration event fired every 500ms. We ran an 8-hour experiment from which the first 1000 network reconfigurations are shown in Figure 3.12(b), marking the percentage of nodes successfully reconfigured at particular transition. On average, 99.91% of network reconfigurations are successful. In another experiment we let the same network reconfigure at the rate of 350ms for 5289 times. In that experiment, 99.68% of nodes successfully follow each network configuration, sending 0.4 messages per configuration transition.

Factors Impacting Reconfiguration. The network-reconfiguration success rate and overhead, which is the number of CM broadcasts and the time reconfiguration delay, depend on multiple-factors: network density, BCP's parameters and the MAC protocols that are scheduled to run at a given configuration. We evaluate the network reconfiguration performance with various MAC protocols: IEEE 802.15.4-complaint CSMA, LPL duty-cycling with a sleep interval of 100ms, a TDMA duty-cycling MAC protocol, and a NULL MAC that transmits without regard to other possible transmissions in the network. In particular, TDMA presents characteristics that match the most challenging aspects that one encounters in the Fennec Fox framework: specifically, the fact that



(a) CM Broadcasts

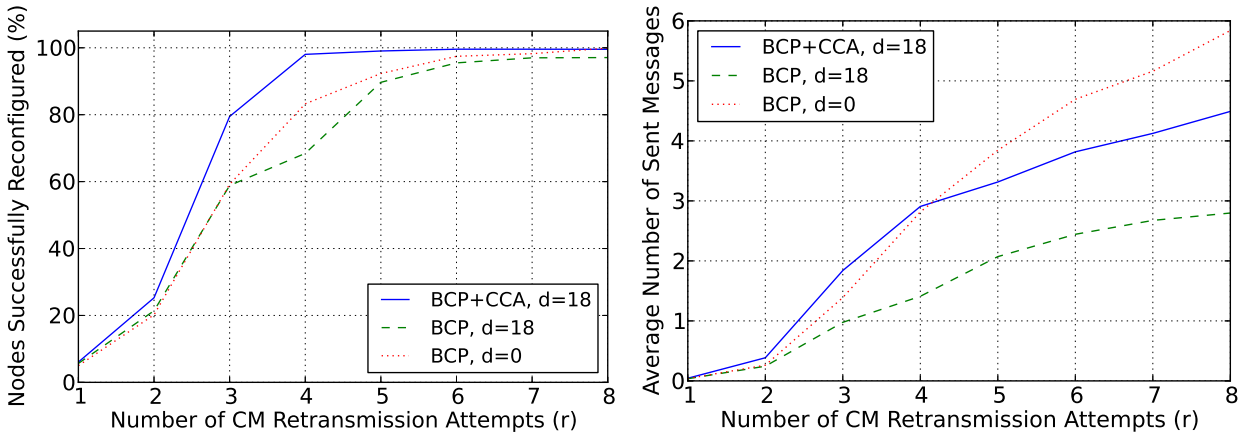
(b) State Transition Delay

Figure 3.13: Radio TX power impact on reconfiguration overhead and delay.

outside the designated time slots packets cannot be received (e.g., CM packets) complicates the operations of the Fennec Fox network-reconfiguration mechanism. Still, in the sequel we present experiments showing how Fennec Fox can handle the presence of the TDMA-style packet timing and, indeed, allow a network to switch between CSMA and TDMA and vice versa.

Network Density. To emulate networks with different densities, we set the CC2420 radio on TelosB motes to transmit at power of $0dB$ to $-15dB$. We found that the reconfiguration is uniformly successful across all the densities, i.e., experiments spanning the entire range of transmission power, except with large values of reconfiguration delay. Network density, however, has a more visible impact on other metrics. Figure 3.13(a) shows that the efficiency of reconfiguration increases at higher density. This is because at higher densities the reconfiguration algorithm suppresses more CM broadcasts in a way similar to the Trickle protocol. Depending on the d value, at the highest density there are 42.1% fewer broadcast transmissions compared to the experiment with the lowest density. Figure 3.13(b) shows that a network with higher density (but the same number of nodes) reconfigures faster. This trend, however, is not observed across all density values.

CM Broadcast delay - d. By delaying the CM broadcast by d ms, we allow nodes to suppress their transmission in case other nodes in the neighborhood are already transmitting the reconfiguration information. As expected, we found that a longer broadcast delay leads to reconfiguration with less CM transmissions. Depending on the density, with a broadcast delay of 10ms, there are on average 0.32 to 0.45 fewer transmissions than with a broadcast delay of 25ms as shown in



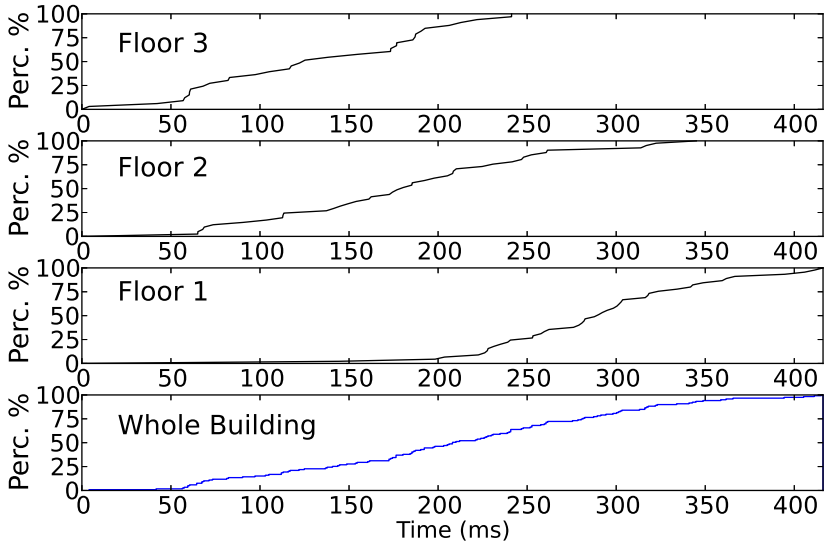
(a) Successful Reconfigurations

(b) CM Broadcasts

Figure 3.14: Reconfiguration from a network with duty-cycling MAC protocol.

Figure 3.13(a). Smaller broadcast delays increase network reconfiguration by as much as 178ms due to the higher probability of transmission collisions. This result, however, is not conclusive for the experiment with the lowest density having lower packet collision probability. The higher CM broadcast rate and the longer reconfiguration time impact positively the success reconfiguration rate, which for d =10ms, 15ms, 20ms and 25ms is on average 99.9%, 99.39%, 98.54%, and 97.94%, respectively. In actual deployments we set $d = 18$. Figure 3.14(a), shows that with $d = 18$ and while using CCA before CM transmission, we can reconfigure almost 100% of the nodes for r greater than 5. With $d = 0$, the success rate is 94-99% for the same range of r. On the other hand, Figure 3.14(b) suggests to avoid CCA and setting $d = 18$ as such CM dissemination consistently requires less CM broadcasts. Because we favor reconfiguration success rate over the reconfiguration transmission overhead, in actual deployments we use CCA.

Impact of Radio Duty-Cycling on the Number of Broadcast Attempts - r. Figure 3.14 shows that to reconfigure WSN between CTP stack and PNP stack, BCP should be set with $r = 5$ and $d = 18$ while using CCA. Without CCA and $d = 0$, setting $r = 8$ successfully reconfigures the network. When $r = 1$ at most 6.06% of the network successfully reconfigures. As r increases, the number of nodes that successfully switch between two configurations also increases. Not surprisingly, as the number of CM broadcast attempts grows, the actual number of radio transmissions grows as well (Figure 3.14(b)): from close to 0 for $r = 0$ to about 2-4 transmissions per



Duty-Cycle	Retry (r)	CM TXs	Delay (ms)
0	1	0.4	80
100	8	6.4	209
200	14	9.2	278
300	15	9.9	387
400	21	13.1	449
500	25	15.1	559

Figure 3.15: LPL impact on network reconfiguration.

node for $r = 6$. In this case, however, larger values of r are still desirable because these additional transmissions contribute to reach a success rate close to 100%.

In many experiments where the radio is always turned on we notice that setting $r = 1$ is sufficient to achieve reconfiguration rate close to 100%. This shows that the presence of other MAC protocols, i.e. MAC protocols that duty-cycle radio operation, has a significant impact on the network reconfiguration. In Figure 3.15 we show how BCP with $d = 18$, $r = 6$ and CCA[8] reconfigures the network across the floors of the testbed building in the presence of a stack configuration with LPL MAC. Examining this graph together with Figure 3.11(a) shows how duty-cycling impacts the latency. With duty-cycling, the complete network reconfiguration takes more than twice as long. Figure 3.15 shows how the network reconfigures progressively, instead of in bursts as in the experiment without LPL. The linear progress of network reconfiguration is the results of radio's LPL with unsynchronized wakeup interval.

We further explore the reconfiguration performance in the presence of LPL MAC with sleep interval beyond 100ms. Figure 3.15 shows the results for the case of a network that switches between two configurations every 5 minutes while using duty-cycling with different sleep intervals.

[8]With those parameters we are optimizing for WSN success reconfiguration rate over the number of CM broadcasts and the WSN reconfiguration delay.

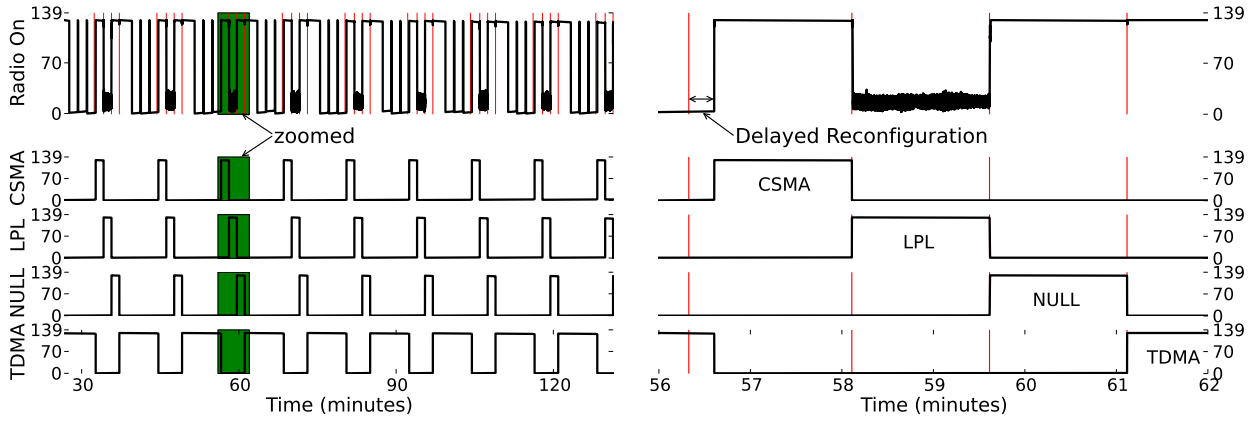


Figure 3.16: Reconfiguration among various MACs.

The table reports the minimum values of r required for high success rate. The results show that longer sleep intervals require more retransmission attempts, up to $r = 25$ when radio periodically sleeps for 500ms. As the number of CP broadcasts attempts grows, the actual transmission count stays around 60-66% of r. As the sleep period increases, the WSN reconfiguration takes longer. The delay more than doubles between 200 and 500ms.

Multiple MAC protocols. The presence of other MAC protocols have a large impact on the performance of WSN reconfiguration. We have already discussed how protocols such as LPL complicate network reconfiguration. Therefore, we conclude the evaluation section with the presentation of an 8-hour experiment, where the network is reconfigured among four network stacks: one using CSMA, one LPL (200ms), one Null MAC, which simply forwards traffic between the network and the radio layer, and the final one, which uses duty-cycling TDMA MAC[9]. Figure 3.16 reports results from the run with the network reconfigured among these four MACs. The left side of the figure shows a sample part of the experiment while the right side shows zoomed-in green section of the left side. The red vertical lines mark moments when events triggering a network reconfiguration take place. The 4-bottom graphs show how many nodes are running with each MAC protocol. Specifically, the upper graph shows how many nodes at the given time have their radio turned on. When CSMA is scheduled to run, all nodes keep their radio on, as expected. When LPL is scheduled, the nodes turn the radio on only periodically and their periods are not

[9]This experiment is performed on the same testbed described in Section 3.2 but after its expansion to 139 nodes.

synchronized. When TDMA is running, the nodes stay on for a while to synchronize their global time and then periodically turn the radio off and on, according to the time schedule.

Figure 3.16 (right) shows how the reconfiguration mechanism handles the situation when the network is in a synchronized sleep mode. The figure starts with all the nodes in the TDMA configuration and their radios turned off. Then, an event occurs that triggers network reconfiguration. Instead, of starting CM broadcast immediately after the event, sending a packet that no other node would receive, the CM broadcast transmission is delayed until the TDMA protocol starts radio again. This delay prevents the broadcasting of CM messages when not a single node would receive CM because the whole network is synchronously shutdown. These experiments show that Fennec Fox can reconfigure a WSN among the four MAC protocols (CSMA, LPL, Null, TDMA), which by themselves could not co-exist in the same WSN.

3.5 Related Work

A major approach to WSN reconfiguration is based on distributing fragments of code that are loaded and executed on the sensor nodes. TinyCubes [143], Enix [25], BASE [74], and ViRe [13] are examples of such systems. Works such as Deluge can perform full-program update on the nodes [87]. Some systems allow users to program WSNs at run-time. Maté [124] allows dissemination of code to be executed in a virtual machine. Tenet [157] allows sending data-flow programs to be executed in the network. Our approach does not send code updates at runtime. Instead, the Fennec Fox framework allows users to specify many applications, each with a dedicated protocol stack, and the conditions under which the WSN self-reconfigures from one application to another.

While previous works show that it is possible to perform incremental or wholesale code updates in the network, these updates must still be disseminated efficiently among the nodes. Efficiency can be achieved by: selecting the subset for update (as in FiGaRo [152]), using a shared infrastructure [187], or minimizing redundant broadcast transmissions as done by Trickle [128]. Fennec Fox uses an approach similar to Trickle to perform efficient network-wide self-reconfiguration.

In the early days of WSN research, most of the protocols were cross-layered, making them difficult to reuse across the applications. Nowadays, there are many sensor network stacks such as Rime [42], uIP [39], and TinyOS IP stack [88] that try to follow the layered protocol model.

Besides these complete stacks, there are now layer-specific protocols (e.g., CTP [62], XMAC [18]) that are designed to allow different protocols at the higher or lower layers of the stack. We leverage the design and implementation of these protocols and provide a framework that allows applications to compose their own stack using protocols of their choice at each layer.

MultiMAC [5] shows that different MACs can be implemented on top of the same radio driver. The pTunes project [215] shows the need for runtime MAC's parameters adjustment and demonstrate it on one protocol. Fennec Fox takes these concepts further by scheduling execution of different MACs that can be initialized with various parameters.

Fennec Fox has been demonstrated to work with other radios than CC2420, i.e. CC1000 [64] and UWB-IR [178].

3.6 Conclusions

We study the problem of executing a set of heterogeneous applications with different communication requirements on a single WSN. Our solution consists in the dynamic self-reconfiguration of the WSN such that it runs the combination of network and MAC protocols that suits well a given application. To do so, we developed the Fennec Fox framework composed of a runtime infrastructure built around a layered protocol stack and a programming language to specify the various WSN configurations and the policy to switch among these in response to various events. Our experimental evaluation showed that our approach can successfully reconfigure a large WSN in few hundreds of milliseconds while incurring little control overhead.

While we demonstrated that is possible to have the WSN self-reconfigure between different MAC protocols such as a CSMA MAC and a TDMA-like MAC, future work needs to address the challenge of switching effectively large WSNs between multiple TDMA protocols which may disable the radio for periods of time that may never overlap. Supporting frequency-hopping and nodes with multiple radios are other research venues to be pursued with Fennec Fox. To truly support a large number of complex applications operating on a single WSN supported with multiple protocols, we need WSN nodes with larger RAM and ROM memories than those available in current mote-class platforms. As more resourceful platforms are already on the horizon, we believe that our approach offers a new path to investigate a new generation of heterogeneous WSN applications.

Chapter 4

System Monitoring

The previous chapter demonstrates the execution of multiple applications on the same WSN. In this chapter, the network runs with applications and system services monitoring the energy-harvesting rate and the rate at which the application's operation consumes power. The challenge in designing and programming such WSN lies in the difficulty of modeling the software and physical dynamics. On the one hand, the system itself is an example of a continuous dynamical system since the amount of the energy stored on the motes in the WSN varies as a function of time. On the other hand, scheduling the execution of tasks has characteristics of a discrete system.

This chapter establishes a connection between the applications and the system tasks operating across the WSN. The system tasks that need to operate continuously are modeled as a feedback control system. The scheduling of all the tasks across the WSN is modeled as a finite state machine. I show that tasks can differ not only in their WSN purpose, executing an application or a system level work but also in the nature of the part of the WSN system that they support. These two types of tasks are demonstrated on an example of an energy neutral WSN system. The implementation of this prototype highlights the need for establishing a data communication channel between the tasks. This communication is necessary to allow the tasks to collaborate and the system tasks to control the execution of the application.

The following work was published at the International Workshop on Energy Neutral Sensing Systems (ENSsys) in 2013 and is a collaboration with Omprakash Gnawali from the University of Houston and my advisor Luca P. Carloni [183].

4.1 Introduction

Energy neutral sensing systems (ENSSys) achieve long-time operation by combining energy-harvesting hardware with software that regulates energy saving and spending. However, simply managing the energy resources is not the goal in itself. These systems have primary responsibility to execute the Wireless Sensor Network (WSN) or Cyber-Physical System (CPS) applications. Thus, there is an open research question on how to design systems running both the target applications, such as sensing and actuating, and the energy-management algorithms that enable long-time execution of these applications.

Despite advances in WSN research [32] and energy-harvesting algorithms [198], there are still few examples of successful technology transfers from research prototypes to actual commercial products. One of the major challenges has been the difficulty of realizing industry-level WSNs that can operate reliably for a long time with minimum energy and maintenance cost while supporting sophisticated applications. To address this problem it is necessary to develop methods for designing and implementing WSN systems that can run effectively both application processes and energy-management processes. Furthermore, these methods should enable design and software reuse across various product deployments.

In this chapter we present the modeling, implementation and evaluation of a WSN running an energy-management system-process and a sensor data collection application-process. We show how the network energy-neutral operation can be modeled as a feedback control system and implemented on the Fennec Fox framework. Using the finite state machine (FSM) model of computation, we show how to separate the execution of the system energy-management from the application. This precludes a potential network communication conflict between these two processes, and it enables designing energy-management algorithms that are application agnostic. We present two case studies to show that the proposed system design and implementation methodology facilitates the composition of complex energy-harvesting systems with improved run-time performance.

4.2 Related Work

Energy-Harvesting. Recent energy-harvesting improvements offer a spectrum of solutions to the problem of battery-constrained WSN life-time [198]. For example, WSN motes can be powered with

solar energy [4; 46; 73] or by people's movement [65]. Motes can also communicate by reflecting TV signals [136] or by harvesting energy from a tiny radioisotope [191]. We apply some of these techniques to enable the execution of sensing applications, which currently operate on non-rechargeable batteries.

Energy Modeling. Modeling energy harvesting and consumption plays a critical role in designing energy-neutral systems. Real-life energy-harvesting traces [154] and power consumption measurements [172] enable the creation of energy models. These models can be used to predict energy-harvesting rates [66] and to run system simulations [169]. We introduce a feedback control model combining both the energy-harvesting and the energy-management algorithms.

Energy-Aware Execution. Energy-neutral systems require careful operation that is constantly aware of its energy resources. The network protocols use energy aware routing [128; 202] and new MAC protocols offer energy-conservation primitives [48; 206]. There are new application development methods for power-efficient sensing [93] and actuating [114]. We show that designing an energy-neutral system should address both the system computation and communication costs.

4.3 Background: Fennec Fox

We implement the energy-neutral sensing system models based on Fennec Fox [182], an open-source framework for the execution of multiple processes in a WSN. The run-time framework execution of multiple processes is specified at design-time by a domain-specific Swift Fox programming language.

Swift Fox programs are used to model the execution of multiple processes across a network of low-power devices as a FSM. Thus, the whole network has a notion of *state*. Each state executes one or more *processes*. Transitions among the network states schedule the execution of different processes on the network motes. This resembles a typical model of an operating system, which switches execution of multiple processes on a single machine.

In Fennec Fox the same set of processes executes on all the motes. These processes require communication services supporting information exchange among the motes. Processes may have different and sometimes conflicting communication requirements. For example, one process may require a duty-cycled many-to-one data collection, and another process may need a high-throughput

one-to-one data streaming [182]. To perform their duties, these two processes use different network and MAC protocols and sometimes different radio configurations. Because these two processes use different communication protocols, they cannot execute on the same network simultaneously. Fennec Fox solves this problem by running these processes asynchronously and reconfiguring the whole network protocol stack.

Each Fennec Fox process is scheduled to execute on a dedicated four-layer protocol stack. The top layer of the protocol stack, called Application, runs the process code. For instance, a process code might periodically sample a sensor, compute the average of the last few samples, and send it across the network. The Network layer runs the network protocol (e.g. CTP [62] for data collection or Trickle [128] for data dissemination) that provides the communication service required by the Application layer code. The MAC layer executes one of the available MAC protocols, such as CSMA, TDMA, and their duty-cycled version. The Radio layer provides the driver code controlling a particular radio chip.

Fennec Fox processes are defined in a Swift Fox program. In the program, each process must specify the application code and list the network, MAC, and radio mechanisms that support the application's execution. The compiled program is installed on the WSN motes. At run-time, as Fennec Fox schedules the execution of a process across the network, it also schedules the execution of the network and MAC protocols together with the radio driver.

4.4 Energy-Neutral System Model

We define a feedback-control model for an energy-neutral sensing system. Within this model we distinguish two types of processes: the application processes and the energy-management processes executing algorithms for sustainable energy-neutral network operation. The energy-management processes generate additional network traffic by reporting the motes' energy harvesting and spending rates. Because the application processes may require network communication protocols conflicting with those used by the energy-management, we propose to asynchronously execute these two types of processes.

Feedback-Control Model of Energy-Neutral System. First, we define the ENSSys parameters. Let $\Delta(t)$ and $\delta(t)$ be the energy consumed by all the motes and a single mote at time

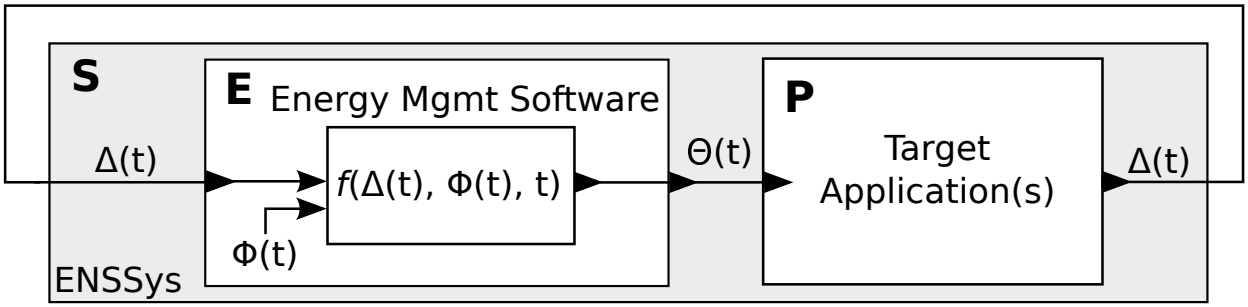


Figure 4.1: Feedback control model of a system executing application and energy-harvesting processes.

t, respectively. Let $\Phi(t)$ and $\phi(t)$ be the energy harvested by all the motes and a single mote, respectively. Both Δ and Φ represent the system continuous dynamics controlled by the energy-management function f:

$$\Theta(t) = f(\Delta(t), \Phi(t), t) \tag{4.1}$$

where $\Theta(t)$ represents the energy-related actuation signals sent to the network and $\theta(t)$ denotes a single control signal sent to one mote. This function provides a general model of the energy-neutral system, whose operation is constrained by the amount of energy consumed and the amount of energy harvested by the motes at a given time.

Second, we define the ENSSys processes. Let P be the set of processes that execute the *target applications* performing sensing, actuating, or computing. Let E be the set of processes with algorithms managing energy-harvesting and ensuring the energy-neutral operation. The whole system consists of S processes, where $S = \{P \cup E\}$. During the ENSSys execution, a subset of these S processes is scheduled to run across the network.

With the defined ENSSys parameters and processes, we compose a feedback control model for sustainable energy-neutral operation. Figure 4.1 shows the energy-management processes E receiving signals $\Phi(t)$ and $\Delta(t)$. The control function f computes the actuation signals $\Theta(t)$ which impact the execution of the target applications P. The execution of the P applications across the network consumes energy at $\Delta(t)$ rate, which is the input parameter to the energy-management processes.

Sensing and Actuation of Energy Harvesting. In the presented feedback-control model, the energy-management processes E require the following two mechanisms:

1. **Energy Harvesting Sensing (EHS):** reports the energy-harvesting rate $\Phi(t)$.

2. **Energy Harvesting Actuation (EHA):** applies the energy control signal $\Theta(t)$ computed by the function f.

These two mechanisms can be implemented as a single or multiple processes. For example, in one energy-management approach, every mote individually monitors its energy resources and adapts its work accordingly [63]. Here, both EHS and EHA may run as a single process. In a different work, motes exchange their energy status [202], which requires separate processes for executing EHS and EHA, because they have different communication requirements for energy sensing and actuating.

Energy Harvesting Communication Requirements. Depending on how and where the ENSSys parameters are computed and sent, the energy-management has one of the following network communication characteristics:

- **Local:** - On each mote individually, EHA computes the actuation signals based on the EHS reports from its own mote. The actuation signals only affect the mote itself.

- **Neighborhood:** - EHS and EHA processes exchange their messages among the motes in the neighborhood. For instance, a mote may broadcast its energy-harvesting status $\phi(t)$ and adapt its operation according to the information $\Phi(t)$ received from other motes.

- **Global-Distributed:** - Both EHS and EHA algorithms run on all the motes and exchange $\Phi(t)$ and $\Delta(t)$ across the network to collectively compute the energy control decisions $\Theta(t)$.

- **Global-Centralized:** - EHS sends both the $\Phi(t)$ and $\Delta(t)$ across the network to the central node. After centrally computing $\Theta(t)$, EHA sends $\theta(t)$ to all the motes in the network.

These different network characteristics of the energy-management processes might be conflicting with the network traffic generated by the target applications. For example, an energy-management process may require CSMA MAC, while the target application may require TDMA MAC. But these two MAC protocols cannot effectively coexist together in the WSN at the same time.

Scheduling Energy Management Processes. To design and implement a single network with the energy-management E and the target application P processes, we propose to schedule their execution asynchronously. This separate execution is mandatory when the energy-management

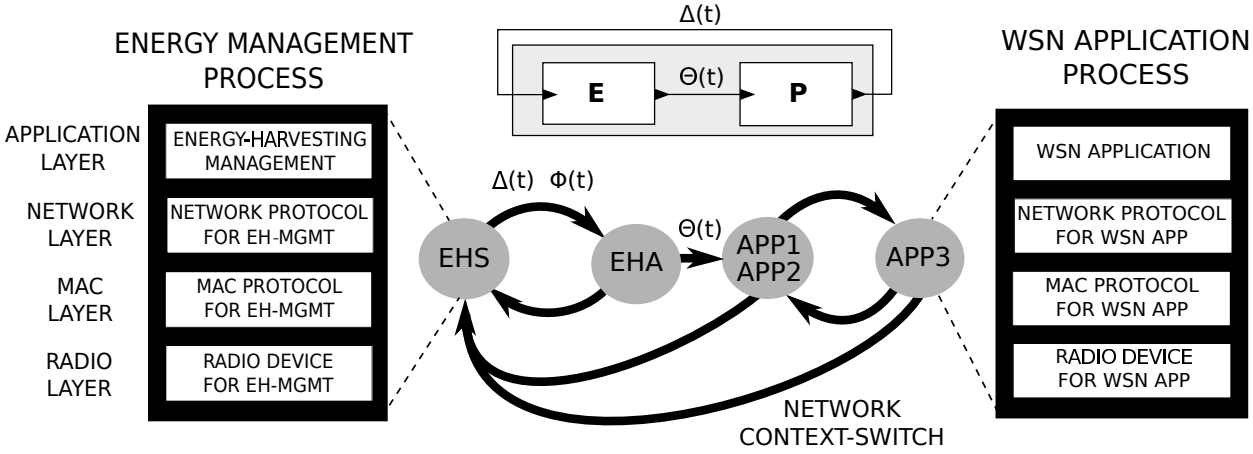


Figure 4.2: WSN context-switch between the application and the energy-management processes.

algorithms and target applications have conflicting communication specifications. When there are no communication conflicts, we still may design ENSSys with the asynchronous execution of the energy-management and the application processes for the following reasons:

1. Preventing degradation of E and P processes execution by isolating their network communication traffic.

2. Improving the energy-harvesting and consumption estimates by computing f without executing the target applications at the same time.

3. Decoupling E from P to enable modular system design and to reuse the E processes with other P.

We apply the FSM model of execution supported by the Fennec Fox framework to asynchronously run the energy-management and target application processes. In the Swift Fox programming language, we map the processes with the conflicting communication requirements into separate FSM states. Then, we create state transition events so that the system only runs either the energy-management processes (and switches the execution among E processes) or runs the target applications (and switches the execution among the P processes).

Figure 4.2 illustrates how we map the feedback control model of the energy-neutral system into a FSM that is programmable in Swift Fox and executable by the Fennec Fox framework. In this

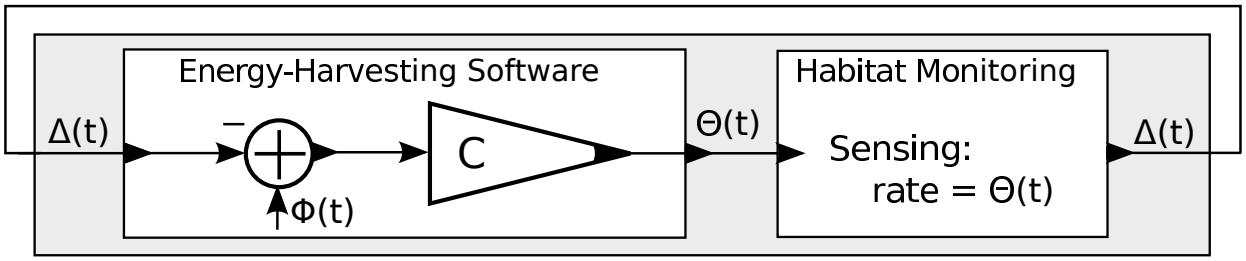


Figure 4.3: Feedback control model where sensing rate is adjusted to the energy-harvesting rate.

example, the FSM has four states: two for EHS and EHA energy-management algorithms and two states for the three target applications denoted as $APP*$. $APP1$ and $APP2$ run concurrently within the same state while $APP3$ runs in a separate state with its own network stack. The transitions from the states running P processes to the states running E processes ensure that the up-to-date control inputs $\Delta(t)$ and $\Phi(t)$ are made available to the system energy controller. The transitions from the states with E processes to the states with P processes set the target applications to run with the most recent energy control signals $\Theta(t)$.

4.5 Case Studies

We present two cases studies to evaluate our approach. The first case study shows the energy-management processes executing asynchronously with the application processes on the same WSN. The second case study applies the existing Fennec Fox mechanisms to achieve run-time system adaptation based on the energy-harvesting rate.

4.5.1 Adapting workload to residual energy

We run a simulated replication of the WSN habitat monitoring study [185]. In this application, sensor nodes were deployed outdoor to collect climate information about sunlight, humidity, air pressure, and temperature. In the original deployment, the key engineering challenge was to set the sensor sampling rate so that the network would operate for a sufficient period of time. We address this problem by using solar cells. In particular, we construct a sensing system where the sensor sampling rate $\Theta(t)$ is dynamically adjusted to the amount of the available energy.

Energy-Harvesting Software. The application sensing rate $\Theta(t)$ is computed by the control function f as the difference between the rate $\Phi(t)$ at which energy is harvested and the rate $\Delta(t)$

at which energy is consumed by the sensing application:

$$\Theta(t) = f(\Delta(t), \Phi(t), t) = C(\Phi(t) - \Delta(t)) \tag{4.2}$$

where C is the signal control scaling parameter. For example, for $C < 1$ the controller sets the application consumption rate below the energy-harvesting rate. Figure 4.3 shows the feedback control model of the WSN for the habitat monitoring. The energy-management function f is computed by the Energy-Harvesting Software, and its output $\Theta(t)$ is sent as a sensing rate parameter to the Habitat Monitoring process.

To implement the modeled system, we specified the communication characteristics of the energy-management software. In particular, we defined the communication mechanisms that deliver the energy harvesting and spending rates as the inputs to the function f, and then transmit the computed sensing rate back to the motes. Specifically, from Section 4.3 we chose to compute the energy-management centrally. Therefore, the network has the Global-Centralized communication characteristics, with:

- EHS collecting information about the energy-harvesting rate $\Phi(t)$ at the central system.
- EHA computing f and disseminating $\Theta(t)$ to all the nodes in the network.

These communication characteristics demand the following network protocols. The EHS many-to-one data collection requires a protocol such as CTP [62], with the $\Phi(t)$ data from all the motes sinking at the central system node. The EHA one-to-many data dissemination needs a protocol such as Trickle [128], with the $\Theta(t)$ data announced by the central system node to all the motes in the network.

Fennec Fox Model and Swift Fox Program. We implemented the habitat monitoring as an application executing on the Fennec Fox framework [182]. To control the application's sensing rate, every 5 minutes the network switched to execute the energy-management processes. Figure 4.4 shows two network states: *Monitoring* - for the execution of the *HabitatMonitor* protocol stack with the *Sense* application, and *EnergyMgmt* - consisting of two protocol stacks running the EHS and EHA software[1].

[1] Although both the EHS and EHA processes require different protocol stacks, they can run concurrently, because the $\Theta(t)$ dissemination occurs after the $\Phi(t)$ data collection, without network communication conflicts.

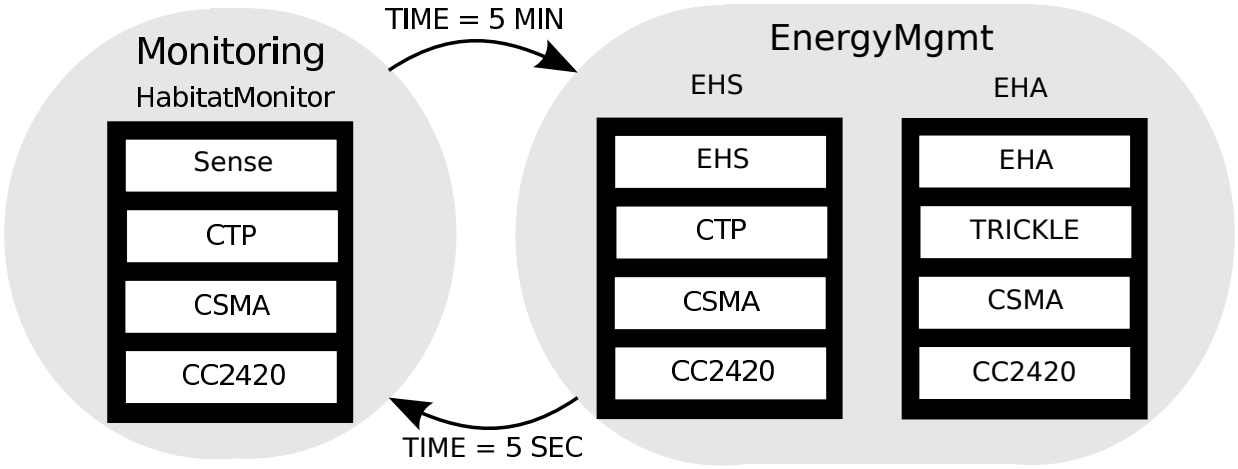


Figure 4.4: Fennec Fox modeling execution of the application and the energy-management processes.

Figure 4.5 shows the Swift Fox program specifying the asynchronous execution of the application and the energy-management processes, together with their supporting communication protocols. We verified the correct execution of the application and the energy-management software by compiling and executing the Swift Fox program on a testbed built with the Open Testbed Framework [184] and running Zolertia Z1 motes. Because those motes do not provide any energy-harvesting capabilities, we used a simulator to conduct WSN energy-management studies.

Simulation. We used the TOSSIM [126] simulator that allowed us to simulate the same code that runs on the target hardware (e.g. Zolertia Z1), with the exception of the radio driver. We extended TOSSIM to support simulation of the solar energy by accessing real-life traces of the irradiance logs from the Humboldt State University in Arcata, California [154]. We developed a simulation interface to configure the energy-harvesting parameters[2] and to define the simulated energy consumption models according to the power consumption reports [172][3].

We compared the performance of the energy adaptive system against three naive energy-management strategies. The first strategy, called Aggressive, runs with a high-fixed sensing rate of 1Hz. The second strategy, Conservative, has a low-fixed rate of one sample every 8 seconds but ensures 24-hour operation. The third strategy has a scheduler, which during the day time hours

[2]In the experiments, we set the solar cell area to 1x2cm, cell efficiency to 20%, and the battery capacity to 400J.

[3]Application sensing and transmitting costs 33mJ, otherwise on average a mote consumes 0.15mJ.

```
1 # Shared variables
2 uint8_t rate = 0

3 # Stack Configurations: conf <conf_d> {<app> <net> <mac> <radio>}
4 conf HabitatMonitor { sense(rate, NODE, 2) ctp(2) csma() cc2420()}
5 conf EHS {ehs(113) ctp(113) csma(0, 1, 1) cc2420()}
6 conf EHA {eha(rate) trickle() csma(0, 1, 1) cc2420()}

7 # States: state <state_id> [priority level] { <conf_id> ... }
8 state Monitoring L3 {HabitatMonitor}
9 state EnergyMgmt L1 {EHS EHA}

10 # Events: event <event_id> {<source> <condition> [scale]}
11 event CheckEnergy {timer = 5 min}
12 event TimeOut = {timer = 5 sec}

13 # Policies: from <state_id> to <state_id> when <event_id>
14 from Monitoring goto EnergyMgmt when CheckEnergy
15 from EnergyMgmt goto Monitoring when TimeOut

16 # Definition of the initial state: start <state_id>
17 start Monitoring
```

Figure 4.5: Program of the system from Figure 4.4.

(10:00-13:10) samples aggressively and otherwise conservatively. Our proposed Adaptive strategy computes the actuation signal $\Theta(t)$ according to the function f from Eq. 4.2, with C = 1.01. This strategy dynamically adapts the sensing rate to the energy-harvesting rate and is programmed according to the Fennec Fox model from Figure 4.4.

Experimental Results. Figure 4.6 shows the experimental results comparing the four energy-management strategies over the same 24-hour irradiance data trace. Table 4.1 reports the aggregate metrics covering the entire experiment. Two metrics of particular interest are the average rate at which the habitat monitoring application attempts to sample sensors and transmit a message, and the percentage of times when at the given sampling rate there is enough energy to run the application.

From the experimental results we observe that the Aggressive strategy spends energy as soon as

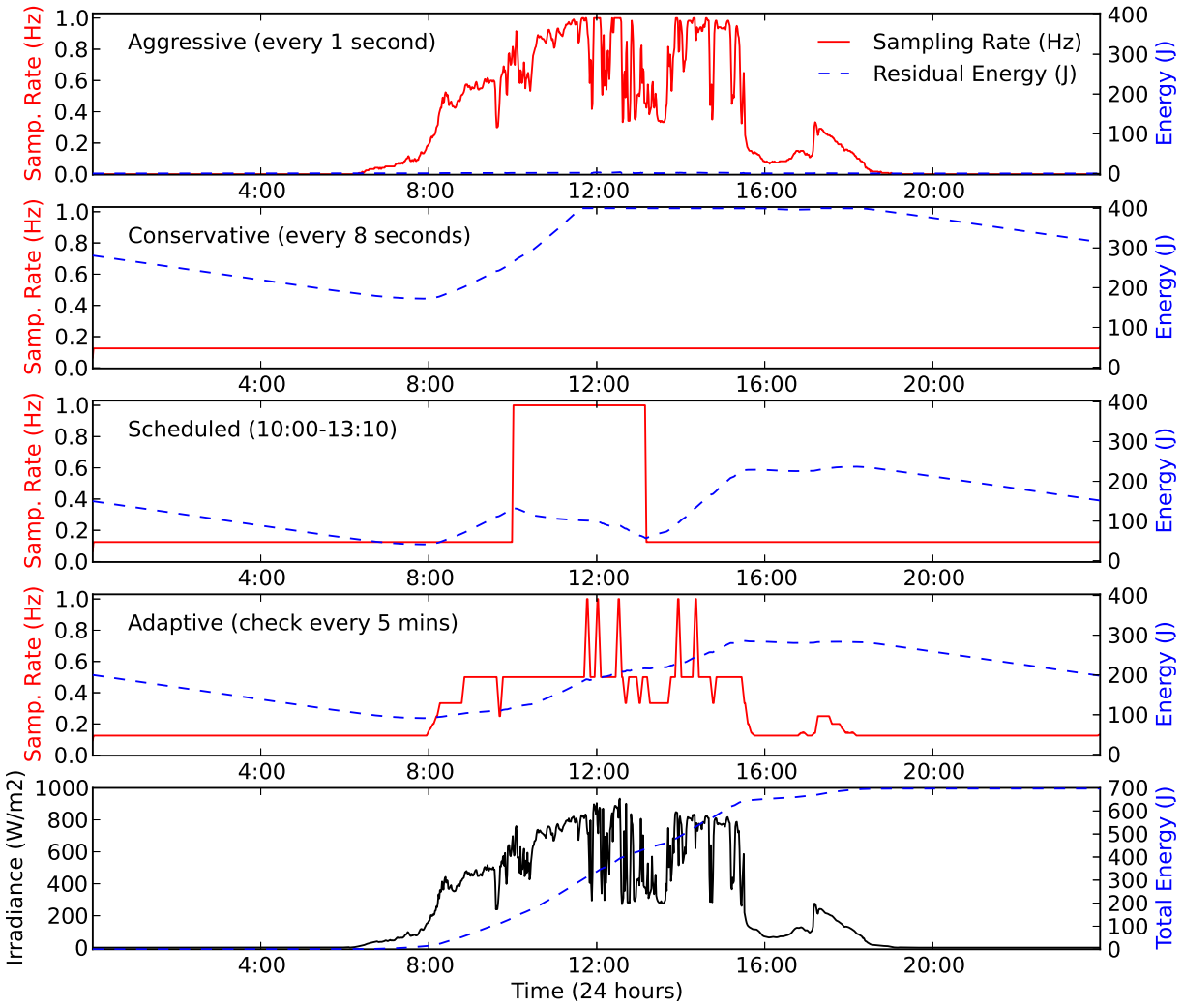


Figure 4.6: Performance of the solar-cell-based energy harvesting and management strategies.

the motes harvest it. Only during the middle day hours the battery gets charged up to 3.3 J. When sampling every second, only 24.3% of the time the motes have sufficient energy resources to complete sensing and transmitting. This yields the average successful sampling rate of 0.243 Hz. In particular, as soon as the day ends, the motes stop working and are not operational until the beginning of the next harvesting period. The Conservative strategy achieves 100% successful sensing and transmitting, with a low 0.125Hz sampling rate. During the middle hours of the day, this strategy is missing the available energy-harvesting resources, because the battery reaches its maximum capacity. The Scheduled strategy operates within the limits of the harvested energy, transmitting

Strategy	Avg. Samp. Rate (Hz)	TotalPackets	Sampling Success (%)
Aggressive	0.243	21006	24.3
Conservative	0.125	10800	100
Scheduled	0.240	20775	100
Adaptive	0.241	20885	100

Table 4.1: Energy-management experimental results.

just 231 less reports than the Aggressive strategy, while successfully operating for all the time. However, to achieve this Scheduled strategy results, we had to run an off-line brute-force parameter optimization to determine when to switch between the two sampling rates. Finally, the Adaptive strategy has a higher average sampling rate than the Scheduled strategy and is transmitting only 0.99% less reports than the Aggressive strategy. Further, the Adaptive strategy sustains a continued sensing and transmitting operation, without requiring off-line schedule optimization and is agnostic to the energy-harvesting rate.

4.5.2 Adapting execution to residual energy

For the second case study, we considered the energy-harvesting active network tags [63]. The tags run on the energy harvested from the indoor-deployed solar cells and transmit messages over a prototype of an ultra-wideband impulse radio. Our goal is to understand how such system would perform with a low-power radio already available on the market. For example, a similar energy-harvesting method was used in the leaf-to-branch communication approach [206], with the CC2420 radio on the Epic Core motes [47]. In the following experiments, we run two applications on the active network tags. The first application is called *Switch*. Once a tag is pressed, this application sends a broadcast message requesting to turn on the light. The second application, *Occupancy*, broadcasts the motion's sensor measurement, every minute.

Energy Spending and Harvesting. We design two simulation experiments with the network tags running the two applications. In the first design, we let both applications to run concurrently until they consume all the available energy resources. In the second design, we prioritize the *Switch* application over the *Occupancy* one and ensure that during the night hours there is enough energy

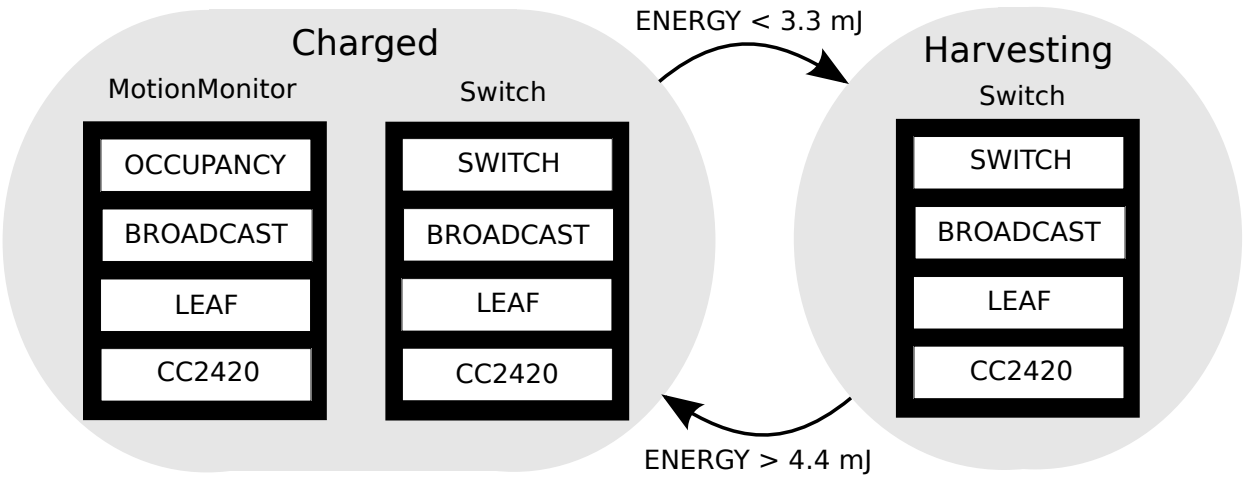


Figure 4.7: Fennec Fox scheduling execution of processes according to the level of the harvested energy.

to press the light switch and turn on the light for at least 10 times.

In the experiment, we use the indoor solar energy-harvesting model [66][4] and derive the indoor irradiance traces from the outdoor ones[5]. Following the energy consumption reports [172], we define the energy-management function f:

$$\theta(t) = f(\delta(t), t) = \begin{cases} 0 & \text{if } \delta(t) < 3.3\ mJ \\ 1 & \text{if } \delta(t) > 4.4\ mJ \end{cases} \tag{4.3}$$

where the output value $\theta(t)$ specifies if there is enough energy resources to run the *Occupancy* application, while securing the energy for 10 *Switch* application executions[6].

Fennec Fox Implementation and Simulation. We use the Fennec Fox energy-based events to trigger a context-switch between the following two states. The *Charged* state runs with the two application processes. The *Harvesting* state only executes the *Switch* application. Figure 4.7 shows the FSM model of the system switching between those two states. Here, the context-switch

[4]The solar cell area is 1x2cm and the cell efficiency is 1%.

[5]Because indoor solar energy levels are 1 to 3 orders of magnitude lower than the outdoor once [66], we divide the outdoor traces [154] by 100.

[6]1 second of sensing consumes 0.015mJ. A message broadcast requires 0.2mJ. A single *Switch* transmission consumes 0.2mJ. Fennec Fox state reconfiguration overhead is 0.19mJ. 1 minute of running *Occupancy* needs 1.1mJ. A tag stops sensing when the energy is less than 3.3mJ. A tag starts sensing when the energy is more than 4.4mJ.

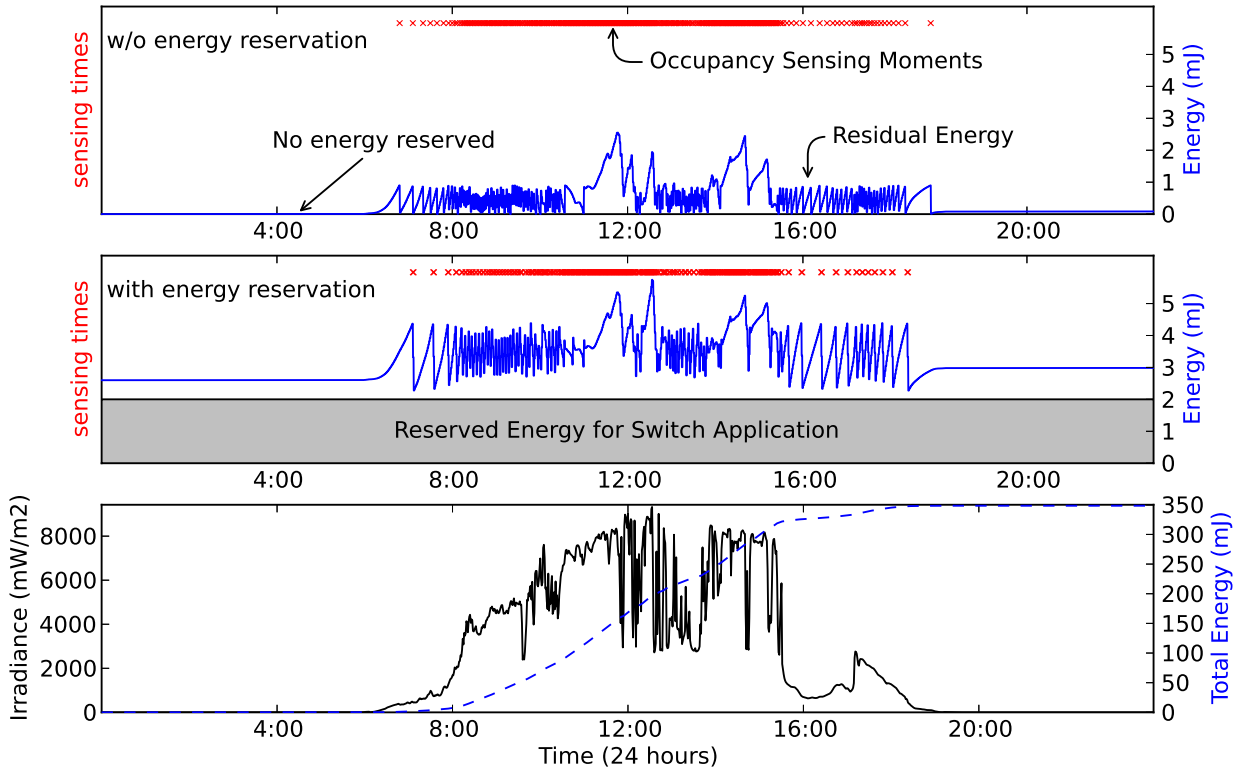


Figure 4.8: Energy reservation for process execution.

is driven by the energy-management function f from Eq. 4.3, thus the system state depends on the residual energy level.

To minimize the broadcast message power consumption, both applications use a low-power MAC protocol. This protocol, called Leaf, follows the leaf-to-branch communication model [206]. Before sending a message, Leaf checks that the radio is turned on and that there are no other ongoing transmissions. Once a message is sent, the radio is turned off until the next transmission.

Figure 4.8 shows the simulation results of the two system designs executing the *Switch* and *Occupancy* applications. In the upper graph, the system concurrently executes both applications. In the middle graph, the system reserves the energy resources for executing the *Switch* application, as in the model from Figure 4.7. The bottom graph reports the irradiance measurements and the available total energy resources.

The experimental results show the tradeoffs between the two system designs. In the upper graph, the *Occupancy* application sent 387 sensor reports between 6:48am and 6:55pm, when it

depleted all the energy resources. In the middle graph, the *Occupancy* application sent, during the 49 minutes shorter period, 356 reports, only 0.92% less than the system without reserving energy for over-the-night *Switch* execution. The system switched between the *Charged* and *Harvesting* states 138 times, with Fennec Fox consuming 27.6mJ.

4.6 Conclusions

We analyze the problem of combining into a single wireless sensor network (WSN) both the energy-harvesting software algorithms and the applications collecting sensor measurements. First, we introduce the models of computation for the energy-management software and for scheduling the execution of multiple processes on the same WSN. The network energy-management is modeled as a feedback control system. The distributed multiprocessing is based on the Fennec Fox finite state machine model of computation.

We demonstrate the system design and implementation methodology on two WSN applications. The first application adjusts the sensing rate according to the rate at which the energy is consumed and harvested. The second example presents two applications with different execution priorities. The lower-priority application runs only when there are enough energy resources to ensure the execution of the high-priority application. These examples show energy-neutral sensing systems with energy managed by following a feedback control model, programmed in Swift Fox and executed by the Fennec Fox framework.

Chapter 5

Installation and Maintenance

The previous two chapters investigate the design-time and run-time challenges with running multiple heterogeneous tasks across the same WSN. The energy-harvesting monitoring case study from Chapter 4 is not only an example of a system monitoring task, but also an example of a WSN system interacting with the physical world. Testing and debugging an interaction of the WSN with physical phenomena requires new tools. This chapter shows how to deploy an experimental testbed in a local environment, thus allowing programmers to observe and understand the conditions that are sensed by the WSN application.

In the context of this dissertation, the work presented in this chapter enables the prototyping of WSN applications with a logic that goes beyond simple data collection. The try-and-error approach in the implementation of various applications interacting with the physical world clarifies the understanding of the WSN nature running with multiple tasks. The research and development experiments conducted on the local testbeds help to recognize the system design patterns and the missing system-level primitives that are needed to simplify WSN programming.

The content of this chapter, which was first presented at the IEEE International Symposium on Industrial Embedded Systems (SIES) in 2013, is joint work with Yong Yang and Dave Cavalcanti from the Philips Research North America and my advisor Luca P. Carloni [184].

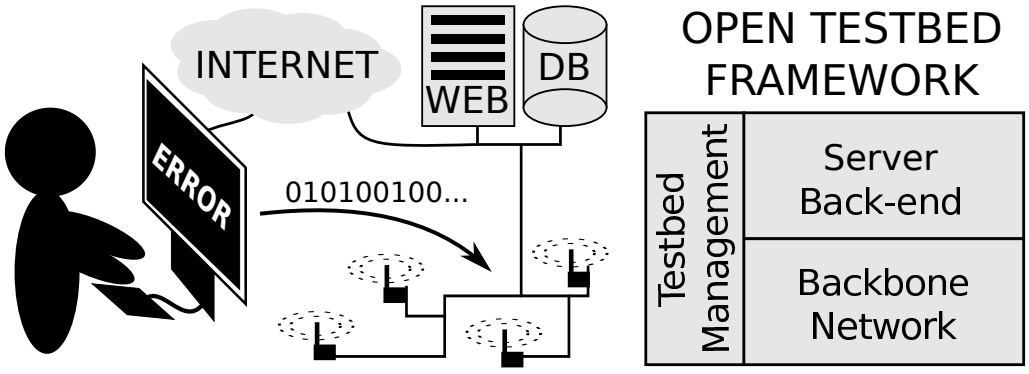
5.1 Introduction

Many cyber-physical system (CPS) [165] applications involve the deployment of low-power wireless networks (LPWN). These networks consist of a set of embedded devices (motes) that combine computation and communication infrastructures with sensing and actuating capabilities to interact with the physical environment. A typical mote has limited computation, communication, and memory resources [94; 163; 217] and minimal operating system support [41; 53; 127]. The interaction of the cyber infrastructure with the physical world is controlled by a distributed, concurrent, and heterogeneous system [32; 179]. The design of such system and the programming of the application software is a complex engineering task [116; 186], which includes a critical validation step [108; 160] that requires physical implementation.

During new system installation, CPS engineers and researchers can find only limited help in the use of network simulators [126; 156] and remote network testbeds [9; 29; 35; 75; 132; 199]. Even the most advanced models of wireless communication and hardware architecture do not offer a testing environment that can capture all the system design aspects that impact reliability and operation. In fact, many issues only become apparent in real deployments. Hence, the correct execution of an application and the evaluation of the scalability and robustness of the networking properties are performed on remote LPWN testbeds. As illustrated in Figure 5.1(a), a LPWN testbed allows an engineer to remotely deploy a new firmware image on every mote through a web-interface. However, due to the increasing demand from development teams and to the small number of LPWN testbeds, there is a limited time to run experiments on them.

Furthermore, neither simulators nor remote testbeds can capture the CPS design peculiarities. These include understanding and processing signals from the actual interaction with the physical world, the intrinsic property of any CPS application. To conduct their research, many CPS developers set up their own *local* testbeds for early prototyping and system evaluation. These testbeds need to provide a good degree of deployment flexibility and support various services for system reconfiguration and experimental evaluation. Given the broad spectrum of CPS applications and the evolution of software during the development process, the testbeds should also accommodate various sensors and actuators deployed on heterogeneous network architectures supporting, for instance, both the IEEE 802.11 (WiFi) and the IEEE 802.15.4 standards.

To design and deploy from scratch a testbed that meets all these criteria is a challenging task



(a) Testbed Control (b) Testbed Framework

Figure 5.1: (a) An engineer tests a program on a remote testbed. (b) High-level structure overview of the framework.

that requires a significant amount of work, which greatly influences productivity and time-to-market. Indeed, *the lack of open tools for creating local LPWN testbeds slows down the progress of CPS research and development.* To address this challenge, we present a framework consisting of a set of tools for rapid deployment of a testbed for CPS applications. As shown in Figure 5.1(b), the framework comprises of three main components:

- the *server back-end*, which stores information about the status of the testbed, collects logging messages sent by the LPWN motes, and provides a web-based interface for remote testbed control;
- the *backbone network*, which connects the installed LPWN with the user-interface, thus allowing the remote control and diagnostics of all the sensor motes;
- the *testbed management unit*, which provides tools and mechanisms for deploying CPS applications, reconfiguring firmware on the LPWN motes, and for monitoring the testbed's performance.

We demonstrate the feasibility of the proposed framework in assisting the development of CPS applications. In particular, we study how well the WiFi-based backbone network can sustain testbed control and high-frequency sensor data collection. We evaluate the network throughput of the testbed's LPWN by providing statistics on how frequently the sensors can be sampled to

collect data over the IEEE 802.15.4 radio. Then, based on examples of sensor events detection in CPS applications, we show tradeoffs between the size of the collected sensor data and the quality of information retrieved from that data. This step is critical to optimize the performance of the CPS application, e.g. in adaptive lighting regulated by traffic sensors - one of our case studies. We illustrate the properties of the framework with two examples of the testbed deployments. The first testbed is installed on the private outdoor parking lot of a commercial building to monitor the occupancy and traffic of cars in this space. The second testbed is deployed inside a university building to experiment with algorithms for people-occupancy estimation.

The proposed framework constitutes a new approach in assisting researchers and engineers with deploying testbeds, which are instrumental for the development of CPS applications relying on wireless communication. It addresses several common challenges. First, it is easy to setup, thus allowing engineers from multiple disciplines to follow the best practices as they quickly deploy *their own and local* CPS testbeds. Second, the flexibility of the WiFi-based backbone network enables the fast testbed re-deployment and node-by-node plug-and-play testbed extension. Third, the relatively low hardware costs, $169 per node, allow researchers in academia and industry to start with a small investment in a few nodes and to quickly obtain a preliminary set of results before deciding to which extent one should augment the deployment.

The rest of the chapter is organized as follows. Section 5.2 discusses related work. Section 5.3 describes the framework's components, while Section 5.4 presents the two case studies. Section 5.5 provides sample testbed statistical information for early stage CPS prototyping.

5.2 Related Work

Over the years two simulators have gained popularity in the LPWN research community: TOSSIM [126] and COOJA [156]. With TOSSIM, it is possible to simulate the execution of software applications written in the nesC language [61] running on a network of motes on top of the TinyOS operating system environment [127] and communicating through the IEEE 802.15.4 wireless standard. Once successfully tested, the same programs can be deployed on any hardware platform supported by TinyOS. With COOJA, it is possible to simulate a wireless network where each mote contains a complete firmware image built for TinyOS and programmed in nesC or built for the Contiki oper-

ating system [41] and programmed in C. Both of these simulators, however, offer limited support to test applications that extensively interact with the physical world through sensors and actuators. In fact, the development of CPS applications and the design of wireless infrastructure to support them cannot prescind from the use of testbeds.

Motelab was one of the first successful LPWN testbeds [199]. Through a web-based interface, an engineer could reserve the network for a few hours, upload a complete firmware image running on every mote, and collect logging messages, which were providing information about the network performance. At first, the testbed setup at Harvard consisted of twenty-six Crossbow Mica2 nodes connected together through Ethernet and the Crossbow MIB600 backbone infrastructure that helped to manage firmware deployment and logs collection. Then, the testbed grew up to 190 Tmote Sky platform nodes, but it is not operating anymore.

Three other LPWN testbeds, which were deployed in a similar fashion as Motelab, are currently available for experiments through remote programming. Deployed at Ohio State University, Kansei consists of over 700 nodes [9]. Through its web-based interface, it allows engineers to run experiments on networks supporting various wireless communication standards, e.g. 802.11, 802.15.4, and 900 MHz Chipcon CC1000 radios, as well as various types of motes, including XSM, TelosB [163], Imote2 and Stargates. Spanning across three floors of a building at TU Berlin, Twist is an indoor testbed that comprises of 204 TelosB motes connected through a network of USB cables to 46-single-board wall-powered NSLU2 computers [75]. It provides a web-based interface for programming and debugging the motes. Finally, Indriya is a testbed with 139 TelosB motes, deployed across three floors of the Computer Science building at the National University of Singapore [35]. With a backbone infrastructure consisting of 6 Mac Mini PCs and a network of USB hubs and cables, Indriya is geographically the largest LPWN testbed, covering an area equal to $23500m^3$.

The above testbeds enable engineers from all over the world to run embedded program prototypes on fairly large LPWNs. The significant number of nodes makes these testbeds particularly suitable to execute communication-oriented experiments, i.e. new network routing and MAC protocols are extensively evaluated on these testbeds to assess their robustness and scalability. CPS researchers and engineers, however, cannot completely evaluate their work on these testbeds because CPS applications require continuous interaction with the physical world. Therefore, during the CPS instrumentation one must have access to the target environment of deployment as well as

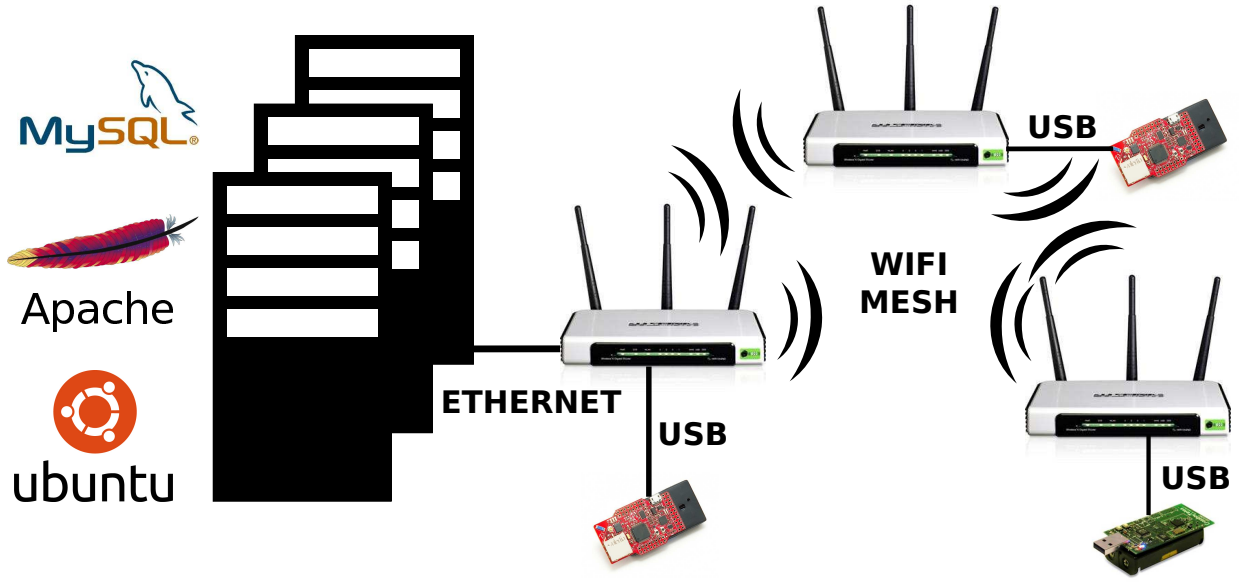


Figure 5.2: General architecture of a testbed that can be deployed with the proposed framework.

to the sensors and actuators. In particular, since a great part of this effort typically involves the positioning, configuring, and fine tuning of motes hosting various sensors and actuators, the direct access to a local testbed is critical. In the following section we present the framework that we built to assist engineers and researchers in the deployment of their own local testbeds supporting the development of CPS applications.

5.3 The Open Testbed Framework

In this section we describe the components of the framework for deployment of CPS testbeds with heterogeneous wireless infrastructures. We briefly discuss how to set up each testbed component and point to online resources for more detailed documentation. All presented software is open-source and publicly available. Specifically, the source code of the presented tools is licensed under GNU General Public Licence, thus allowing everyone to freely use and modify the programs.

Figure 5.2 shows the complete architecture of a testbed that can be deployed and controlled with our framework. A collection of various LPWN motes, such as TelosB or Z1 motes, is controlled through a backbone network of WiFi routers from a server back-end. The server and the WiFi

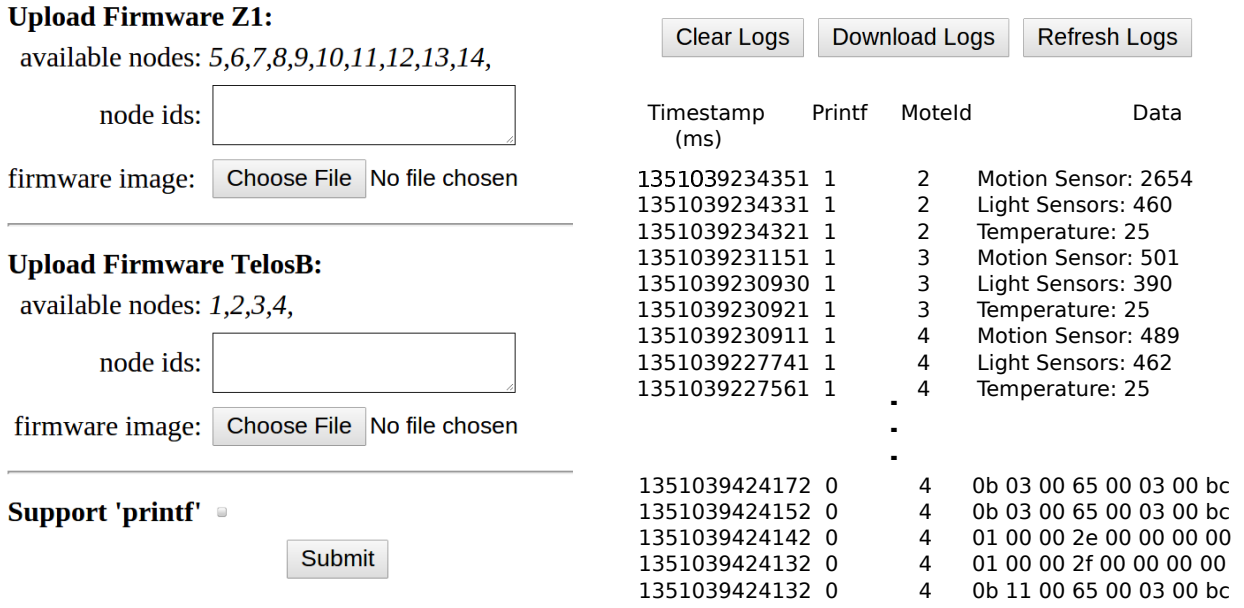
routers can be connected either through a private network or the Internet. The testbed management software running on the server provides a user interface and a set of mechanisms to configure the testbed according to the characteristics of the particular applications. Each LPWN mote, which can host a particular set of sensors and actuators, is connected to a router through a USB cable. The application software and the operating system firmware running on the motes can be efficiently uploaded, tested, and configured through the backbone network.

The presence of the wireless backbone network is a distinctive element of our approach that significantly increases design productivity. Once the development phase is completed and the specific CPS is ready to be released and produced, the backbone network can either be removed or scaled down as appropriate.

5.3.1 Server Back-End

The server is setup with the Ubuntu operating system. In our current deployment, we have configured it with the 32-bit Ubuntu Server 12.04 LTS, which is a long-term release with support guaranteed by Canonical Ltd. for five years, starting from April 2012. The back-end includes also a database server (MySQL, version 5.5.24) and a web server (Apache, version 2.2.22). The database server stores information about the testbed configuration and debugging messages collected from the LPWN motes. Depending on the size of the data and the data processing algorithms, the sensor measurements are either saved in the same MySQL database, in a local file on the server, or are sent to Hadoop for distributed processing.

Figure 5.3 shows two screen-shots of the web-interface running on the server. Figure 5.3(a) shows the firmware uploading interface. From the list of *available nodes*, the user can specify the IDs of those nodes that should be reprogrammed. In the *firmware image* field, the user can choose the location on a computer where a new firmware for a given mote architecture is be stored. Before starting the experiment, the user tells the framework the format in which the logging statements are sent from the program under design. A log message can be either a plain text statement or a byte-encoded report. After reconfiguring the testbed and starting an experiment with the new firmware, the user-interface switches to reporting online logging messages, as shown in Figure 5.3(b). The figure shows two different experiment runs: one set of logs is sent in form of plain ASCII text messages and the other is encoded as a sequence of bytes. Each log statement starts with a

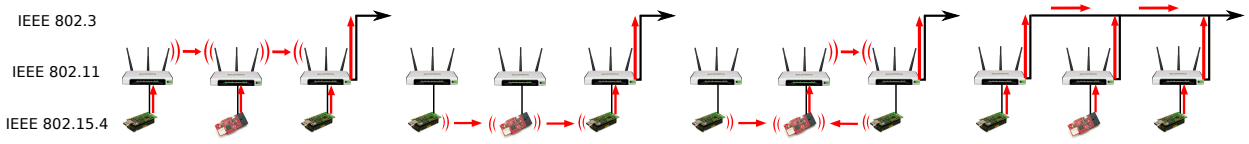


(a) Upload Interface (b) Sample Logs

Figure 5.3: Screen-shots of the framework's user interface for new firmware installation and downloading logs.

timestamp that is marked by a WiFi router when it receives a log message from the corresponding mote attached through the USB connection. A copy of the logs can be downloaded from the server into a local computer. from the server.

A complete server configuration, including the operating system, database, and web-interface, can be downloaded as a virtual machine image.The virtual machine format allows us to deploy the server image on most computers and cloud systems. For example, the server can be easily started through VMware Player or VMware Workstation 9.0 on a Windows or Linux PC, or through Fusion 5.0 on a OS X computer. Alternatively, it can be started as a virtual server on an ESX Server cloud infrastructure. Once the server is running, it can be accessed through its IP address. Hence, by simply typing the server's IP address in the URL field of a web-browser, the user can see the welcome page of the framework.



(a) WiFi (b) Mote (c) WiFi-Mote (d) Ethernet

Figure 5.4: The sensor data collection experiment configurations on the framework built with heterogeneous network standards.

5.3.2 Backbone Network

The testbed backbone network connects together the server and all the WiFi routers. The routers can be attached to a wired network or form a wireless mesh network with other routers. Furthermore, the backbone network can be setup as a combination of both.

When the routers are connected to the wired network, the server communicates directly with each router through its assigned IP address. When routers together create the ad-hoc WiFi mesh network, the server reaches a particular router through a multi-hop path across other routers and a gateway router: in this case, at least one router is attached to either a private network or the Internet. Hence, at a minimum, the backbone network requires only one router to be connected to the server back-end. Since the backbone network serves as the ad-hoc mesh network, the testbed can be efficiently re-deployed in various places and new routers can be seamlessly merged with other routers' network. The routers, however, need to be connected to a power source. Nevertheless, the testbed is simpler and faster to deploy than most existing LPWN testbeds because the routers do not require any form of additional wired connection.

The WiFi routers are running the OpenWrt Linux-based operating system. The OpenWrt project[1] provides tools to build complete firmware images for various WiFi routers and for multiple embedded hardware architectures. The project combines a Linux kernel with a set of tools that are commonly used in wireless networking. In particular, the framework is setup with the Optimized Link State Routing (OLSR) protocol to organize the ad-hoc mesh network among the WiFi routers. Each router is time-synchronized through the Network Time Protocol.

Support for Heterogeneous Network Architectures. The existing testbed architectures

[1]OpenWrt Project: `http://openwrt.org`

preclude experimenting with various wireless networks because they are solely based on a wired backbone. While the WiFi routers provide wireless communication according to the IEEE 802.11 standard, motes such as TelosB and Z1 offer wireless communication based on the IEEE 802.15.4 standard. With such communication heterogeneity, an engineer can test a program in a network solely operating within the IEEE 802.11 domain, utilizing only the IEEE 802.15.4 communication, or with combination of the two protocols. Thus, our proposed WiFi-based backbone network architecture allows us to evaluate embedded software in multiple configurations of wireless network standards.

Figure 5.4 shows four configurations of wireless networks supported by the framework. Figure 5.4(a) shows a configuration where all sensor data are sent from each mote through USB directly to the attached WiFi router, and then the data are routed to the server through WiFi mesh network. Figure 5.4(b) shows a configuration of the testbed with one mote collecting sensor data from other motes and sending the collected data to the attached WiFi routers that forward the data to the server. Figure 5.4(c) show one of the possible combinations of the testbed configurations, where multiple motes collect sensor data from other motes. The collected data are sent to the attached WiFi router and further to the server through the WiFi mesh network. By attaching each router directly to Ethernet, as shown in Figure 5.4(d), the testbed operates in the same fashion as existing wired-based testbeds. As we will show in Section 5.5, for CPS development a testbed operating only on the wireless backbone is as good as the testbed relying on the wired backbone.

The network architecture configuration for a given experiment depends on the application scenario. For example, consider the task of collecting sensor measurements for a CPS application. An engineer might be required to test an embedded program for various sensors, some connected through wires and others employing wireless communication (IEEE 802.11 or IEEE 802.15.4). As shown in Figure 5.4(a), to experiment with CPS sending sensor measurements through WiFi network only, the testbed needs to be configured with every mote sending data over the USB connection to the attached router. For many CPS applications, however, engineers do not have the luxury of deploying a wired network for sensor-data collection. In some applications, even WiFi is only allowed for testing and debugging, but not in the final version of the system deployment. Hence, it is important to evaluate a CPS prototype within the restricted domain of LPWN, as shown in Figure 5.4(b).

During the CPS design-time, an embedded software engineer is responsible for building a firmware consisting of a sensor data collection application supported by a multi-hop ad-hoc network routing among the motes. The sensor measurements are routed to a collector using network protocol such as the Collection Tree Protocol (CTP) [62], which runs on top of the IEEE 802.15.4 MAC protocol. The collector further forwards the sensor data to a database or a program parsing sensor data logs. In some applications, however, where the number of sensor motes is large, the LPWN itself may not sustain the whole network traffic (see CTP study in Section 5.5). Then, one may want to consider a CPS implementation with a two-tier network. At the lower-tier, the ad-hoc mesh network is setup among one or more collectors. At the higher-tier, the collectors are connected together through a network consisting of the WiFi routers, as shown in Figure 5.4(c).

Deployment Flexibility. The flexible framework architecture allows us to adjust the placement of the LPWN motes during the CPS design and prototype evaluation phase. Once the target application is defined, it is necessary to establish where and how many sensors and actuators should be installed. At the beginning, the blue-print of a new cyber architecture is guided by intuition and experience. Later on, through the iterations of experiments on the CPS deployment on an actual testbed, the cyber architecture becomes more precisely characterized until the number and position of the motes with the specific sensor and actuators is completely determined. During these adjustments, the flexible testbed architecture enables engineers to reorganize the placement of the nodes and to attach more nodes where needed.

The proposed framework simplifies the task of finding critical parameters of the system under design. For instance, the design of a new CPS application requires to specify how often a sensor should sample a given physical entity. This question is usually difficult to answer. On the one hand, higher frequency sampling rates provide more data about the surrounding environment. On the other hand, low-power embedded wireless devices have limited bandwidth and are constrained by power resources that are mostly consumed by collecting sensors' measurements and transmitting data over the radio. Therefore, each CPS deployment must find the right balance for the particular system implementation. One needs to find the minimum sampling frequency that guarantees correct interaction with the physical world and the maximum sampling frequency that the cyber infrastructure can maintain (Section 5.5). The heterogeneous testbed architecture that we propose helps define the CPS sampling parameters. A CPS architect can first rely only on the WiFi net-

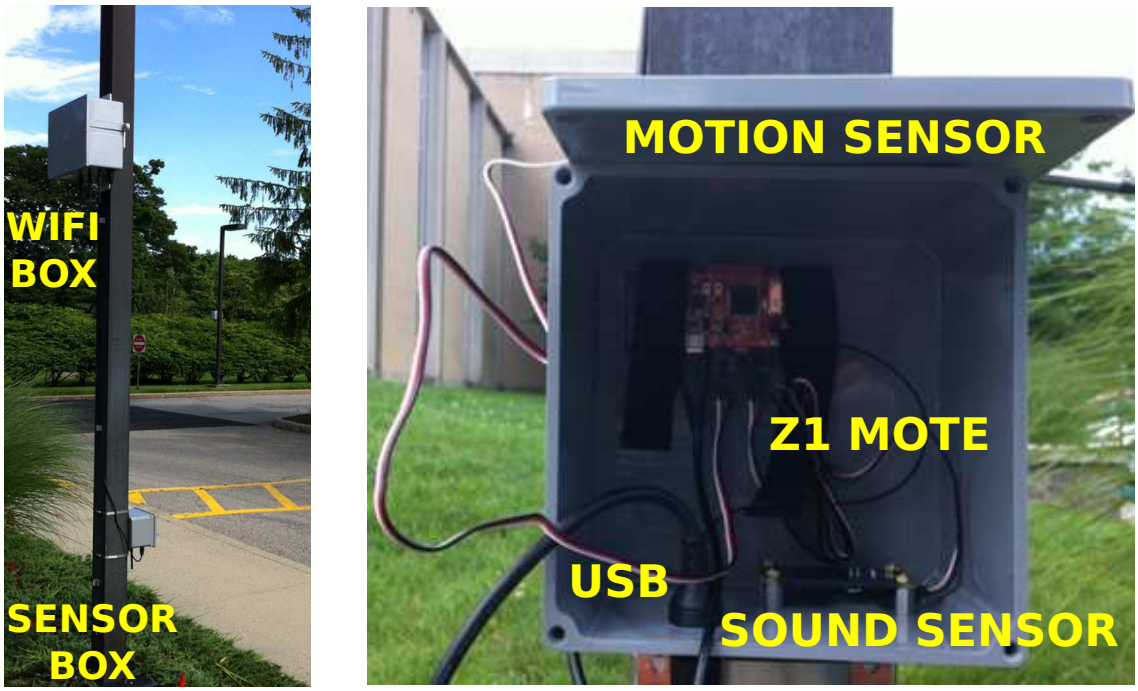
work collecting sensor samples at a higher rate than the LPWN can sustain. Then, by studying the collected sensor samples, the lower sampling frequencies can be identified together with the values of the system parameters impacting its responsiveness, correctness, and lifetime. Once the new sampling frequency parameters are determined, the CPS application software can be migrated into the motes architecture and validated using the same testbed deployment.

5.3.3 Testbed Management Unit

Our framework provides a set of tools connecting together the third-party software running on the WiFi routers and the back-end server. Software programs running on each router manage the interaction between the motes and the back-end server. When a new mote firmware image is being uploaded from the server, each router flushes the firmware into the mote's program Flash memory and reports back to the server the status of the mote. The router receives logs messages from the program executing on the mote and stores them locally in its own memory before they are downloaded into the server. The tools operating on the WiFi routers are compatible with the Open PacKaGe Management (opkg), a lightweight package management system for embedded Linux devices. This software-distribution format permits installation and system updating over the Internet without interrupting the testbed services.

The framework tools running on the server offer a web-based user interface to communicate with the routers. Particularly, this interface comprises of two programs. When a user uploads a new firmware through the web-interface, the first program checks the correctness of the user's input, verifies network connectivity with the WiFi routers, and initiates a secure SSH-encrypted connection with each router. The secure connection with the WiFi routers increases the safety of the intellectual property of the software and the privacy of the data containing sensor measurements and actuator control signals. While an experiment is running, the second program periodically checks the status of the testbed and informs the user when a misbehavior is observed. Every minute, the testbed server securely downloads the log messages from the motes which have been buffered on each router. The logs are stored in the database server and the latest log update is continuously displayed on the user interface. The source code of the testbed OpenWrt packages and the source code of the set of the tools running on the server are available online[2].

[2]Project Repository: `https://github.com/mszczodrak/otf`



(a) Light Pole (b) Inside Sensor Box

Figure 5.5: Testbed deployment on a parking lot.

5.4 Testbed Deployment Examples

In this section we present two case studies of actual deployments made by using the proposed framework: the first example is an outdoor testbed deployment in a commercial environment while the second example is an indoor testbed deployment in a university building.

5.4.1 Outdoor Parking Lot Testbed

We deployed an outdoor testbed in the parking lot at Philips Research North America in Briarcliff Manor, New York. Currently, fourteen light poles, spanning an area of 80x100 meters, are instrumented with the testbed hardware. The testbed is used to evaluate prototypes of *Intelligent Outdoor Lighting Control* applications. These applications focus on detecting traffic (e.g., vehicles and pedestrians) and actuating on the system composed of the outdoor lighting network to improve energy efficiency and to meet safety and user requirements. For instance, our application allows autonomous light-dimming based on the presence of people or on the movement of cars.

Hardware Infrastructure. Figure 5.5(a) shows the mounting of the the testbed hardware on one of the light poles. Each pole comprises of one WiFi box and one sensor box. Each WiFi box contains a TP-LINK 1043ND WiFi router with a 400MHz microcontroller and 8MB of Flash and 32MB of RAM memories. The router is secured inside a plastic box. To maintain strong WiFi signal reception, all three router's antennas are extended outside of the box. To create the USB connection with motes, the router's USB port is extended outside of the box. In total, three industrial USB cables are used in order to decouple the WiFi box from the sensor box, for installation and maintenance purposes. Each WiFi box draws AC power from the light pole.

Figure 5.5(b) shows an opened sensor box. The box contains a Zolertia Z1 [217] mote, connected to different sensors depending on the particular application. For example, in this figure the Z1 mote is connected to a sound sensor mounted at the bottom of the box and to a motion sensor monitoring the street through a secured hole in the front cover of the box. The sensor box has attached a USB cable that is connected to the Z1 mote and with the WiFi box. Each sensor box is powered through USB.

The WiFi box and the sensor box are assembled out of commercial off-the-shelf components. Each box is professionally assembled and tightly sealed to protect the electronic devices from water damage. The WiFi boxes are framed with metal blades, allowing us to screw each box into a light pole. The sensor boxes have metal stripes for an installation on various poles. The testbed has been running for over a year and it has survived diverse extreme weather conditions.

Table 5.1 lists the testbed hardware and its cost[3] per node deployed outdoor. Each testbed node is assembled with three units of antenna extension cables and one unit of the rest of the items listed in the table. The total cost of a single node deployed outdoor is approximately $282, with $113 spent to secure the electronic hardware equipment. However, the actual total cost of a single light pole instrumentation is higher due to the labor of the technicians preparing the boxes and soldering the external antenna, and the electricians mounting the boxes and connecting them to the power source out of each light pole. Despite that, the costs of our testbed quickly pay back in terms of increased productivity.

Software Infrastructure. The testbed server is deployed on a private cloud infrastructure. The OpenWrt embedded Linux operating system is installed on all the WiFi routers. In particular,

[3]Price as of March 5, 2013

Name	Description	Price	Indoor	Outdoor
WiFi Router	TP-LINK WR1043ND with Atheros AR9132 400MHz CPU, 8MB Flash, 32MB RAM, 4 Gigabit ports and one USB.	$52.95	x	x
Z1 platform	Zolertia Z1 with TI msp430 microcontroller, 92BK Flash, 8KB RAM and Phidget ports.	85.00€	x	x
Antenna Cable	RP-SMA Plug to RP-SMA Pigtail 19" (x3)	$14.99		3x
Antenna	2dBi Gain Antenna with U.FL	$4.49		x
USB	A Male to Micro B, 6ft	$5.99	x	
USB	Waterproof USB Cable-A to Mini-B 78"	$16.50		x
USB	USB Mini-B Waterproof Mountable 20"	$16.91		x
USB	Waterproof A Female to A Male 20"	$17.43		x
Box	12x12x4 (inches) - Cantex 5133714	$36.89		x
Box	6x6x4 (inches) - Carlon E987R	$11.78		x
Indoor Cost per Node				**$169**
Outdoor Cost per Node				**$282**

Table 5.1: Outdoor and indoor testbed hardware and cost.

the routers are operating on the OpenWrt Backfire 10.03.1 stable release with Linux kernel 2.6.32. When powered-on, the routers automatically configure the ad-hoc mesh network through the OLSR protocol. One router is placed inside a building and serves as the gateway to other routers mounted on the light poles.

Research and Development Practice. The presented framework enables us to prototype embedded software running on the LPWN and to enhance lighting control performance by testing various system configurations. The framework offers three key advantages. First, the WiFi-based flexible backbone network architecture represents the least invasive testbed deployment approach for a commercial outdoor environment. Whereas any wire-based approach would require laying down wires across the street and parking lot, the wireless solution only needs a connection to a

power-source which in most industrial environments can be found in proximity. In the deployments of the outdoor CPS applications, the light-pole is a good infrastructure to connect to a power source. Second, thanks to our framework we can remotely update the firmware of all fourteen Z1 motes within less than thirty seconds. Without the framework, updating firmware of just one mote would take over ten minutes because it would be necessary to go to the field, open the enclosure, and connect it to the development computer. Third, the online logs that are gathered from the firmware running on the Z1 motes, enable continuous debugging of the sensing firmware while collecting actual sensor measurements, and provide quick feedback about the system performance.

5.4.2 Indoor Office Testbed

We deployed an indoor testbed at the Computer Science Department of Columbia University. The testbed spans an area of 10x18 meters, placed across labs and offices of one floor.

Hardware Infrastructure. We use two models of the WiFi routers: TP-LINK TL-WR1043ND and TP-LINK TL-WDR4300. Out of 16 routers, 2 are mounted far from any power source and, therefore, are powered through Power-over-Ethernet (PoE), following the IEEE 802.3af standard. During the experiments these routers are not using Ethernet, so the WiFi mesh network is supported with only one gateway node. One of the routers is connected to two motes. The routers are installed with the OpenWrt Attitude Adjustment 12.09 stable release with Linux kernel 3.3.8. The routers are connected to 17 motes: 4 are TelosB and 13 are Zolertia Z1.

The testbed server was first deployed as a virtual machine running on a laptop computer and then migrated to the department IT cloud, where it has assigned a unique IP address and DNS record: this allows us to connect to the server through its own URL address. The virtual machine is configured with a single-core 1GHz processor and 1GB of RAM.

All seventeen motes create a LPWN collecting sensor measurements, which are then stored in a database and processed as part of CPS applications for smart-buildings, such as room-environment monitoring and people-occupancy estimation. The TelosB motes gather information on temperature, humidity, and light through a set of integrated sensors. The Z1 motes are factory-assembled with a 3-axis digital accelerometer and a low-power digital temperature sensor. In addition to these sensors, each Z1 mote is connected to two Phidget sensors, which can provide the following sensing capabilities depending on the given application: touch, distance, infrared reflective, sound,

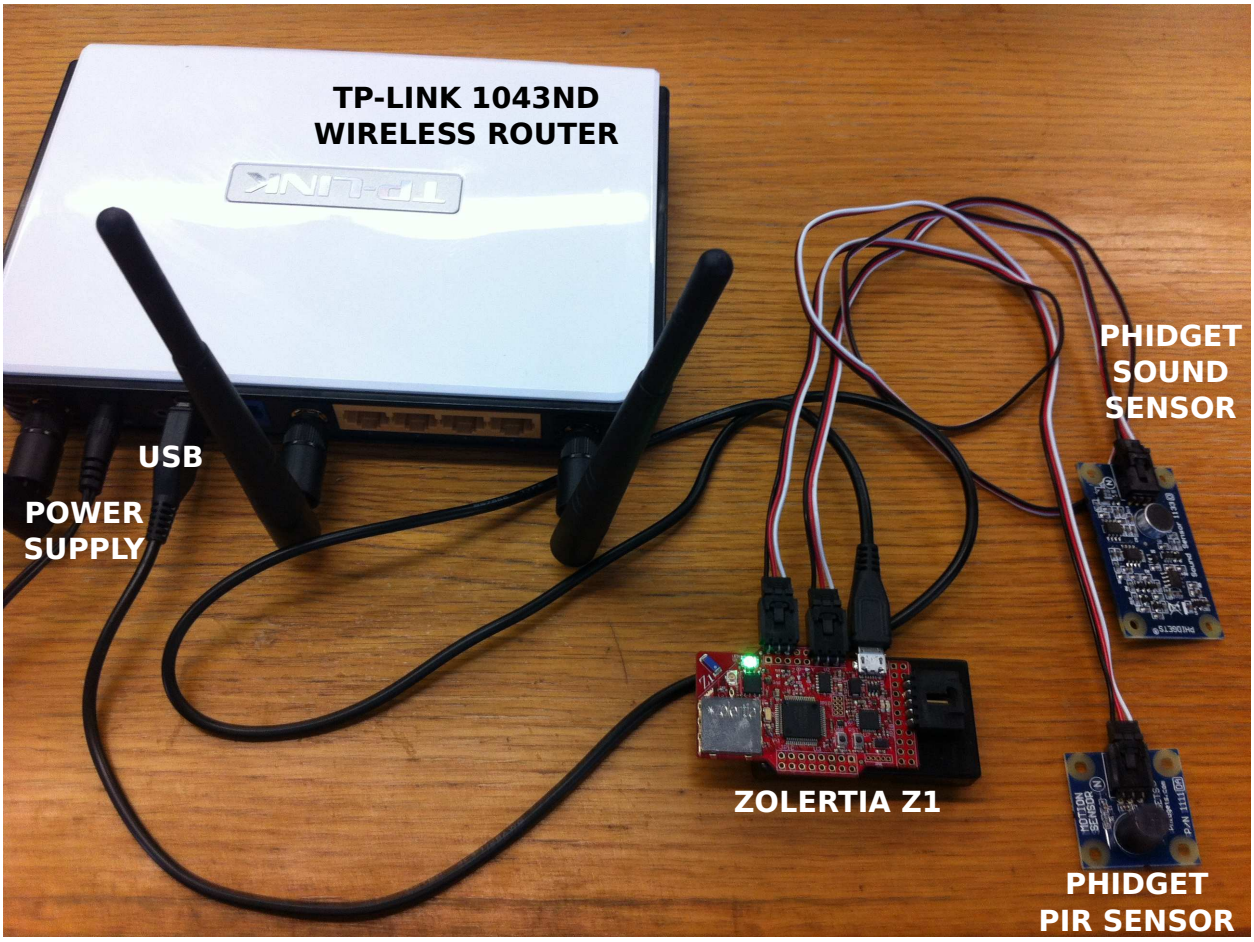


Figure 5.6: An indoor testbed node assembled with the TP-Link 1043ND WiFi router, Zolertia Z1 mote, and two Phidget sensors.

vibration, passive infrared motion, magnetic, thin force, and precision light.

Figure 5.6 shows one of the deployed testbed nodes: the TP-LINK 1043ND WiFi router is connected to the power source and to the Z1 mote through a USB cable. The Z1 mote is connected to two sensors, PIR (motion detection), through the available Phidget ports. Combined, the assembly of one testbed node and the uploading of the OpenWrt firmware takes approximately five minutes. As part of the node-installation process, each mote's corresponding router is connected to a power source and the sensors' placement and orientation are adjusted. When a node is turned on, it automatically becomes part of the testbed network. The wireless backbone network and the firmware-upgrade capabilities allow users to remotely control the testbed without interrupting the

work of people who are present in the area of deployment.

The hardware and its costs per single testbed node deployed indoor are reported in Table 5.1. The complete testbed-installation cost depends on the number of nodes and the price for deploying the testbed's virtual server. A single node, as the one shown in Figure 5.6, costs $249 ($169 without the sensors). Depending on their quality, sensors and actuators cost in a range of $0.99-$45.00 per item.

Software Infrastructure. The motes run applications that are developed using the Swift Fox programming language on top of the Fennec Fox framework [182] and TinyOS [127]. Combined, these provide the necessary software support for configuring the LPWN multi-hop message routing and for designing applications interacting with sensors and actuators.

Research and Development Practice. The indoor testbed is used for research and development of smart-building applications and for educational purposes to allow students to acquire hands-on experience with these hardware and software. The deployed sensor network is currently collecting sensor measurements for occupancy-estimation applications in commercial buildings. Our framework effectively supports CPS research by providing the following advantages. First, the remote firmware reconfiguration enables us to install various embedded programs without interrupting the work of people occupying the space under monitoring. Second, because WiFi routers require only a single wire, either a power-cable or PoE, testbed installation becomes more flexible. Wiring additional cables would increase deployment cost and might depend on obtaining permits, which would delay the testbed deployment. Third, the flexible testbed infrastructure makes it possible to quickly move sensors around the building as we look for the most appropriate places for gathering sensor measurements and for monitoring the areas of the highest interest. This is particularly important in establishing ground truths for the development of event-detection algorithms.

The heterogeneous wireless backbone network allows us to proceed with the CPS deployment in two steps. In the first step, the firmware running on the motes sends data over the USB to the attached router which forwards messages over the WiFi to the testbed's server. The high bandwidth of the WiFi routers enables us not only to collect enough data to establish ground truths but also to determine the key parameters that influence the results of the interaction with the physical world. After studying the environment, in the second step, we ran the same embedded program as in the first step, but this time we used LPWN's multi-hop communication, instead of sending

messages over the USB and WiFi routers. Recompiling the firmware to use LPWN instead of the USB connection and installing it across all the LPWN motes takes less than a minute. Setting the communication type parameter (USB or wireless) and tuning the system performance parameters is as simple as changing their corresponding values in the Swift Fox [182] program that configures the embedded firmware.

5.5 Testbed Evaluation

In this section we present an evaluation of the two testbed deployments introduced in Section 5.4, and on these examples we show how to implement a testbed for CPS prototyping. First, we show how much sensor data can be collected by a single mote. Then, we present the exemplary performance of the WiFi mesh network throughput measured while routing the sensors' data in indoor and outdoor deployments. These experiments confirm that the WiFi-based testbed's backbone network is sufficient to collect sensors' samples. After studying WiFi throughput, we show measurements of the LPWN network throughput. These experiments indicate the data rates that can be sustained by IEEE 802.11 and IEEE 802.15.4 standards. Finally, we present examples of CPS instrumentation that finds the sensor sampling frequency necessary to detect an event. Based on motion and distance sensor data traces, we provide empirical results on how frequently these sensors need to gather samples to support applications such as occupancy estimation and parking movement detection.

Maximum Sensor Sampling Frequency. We start the evaluation by asking how the framework helps us better understand the environment which the CPS application interacts with. This problem comes from the questions that often arise at the beginning of many CPS developments: how the events of interest look like, how much data is necessary to detect an event, what is the sensor frequency sampling, and how often the sensor samples should be collected. To answer these questions, CPS engineers start with collecting as much data as possible. Therefore, it is crucial to estimate how much sensor data a single mote can generate.

The maximum rate at which a mote can collect the sensor samples is limited by the mote's architecture, i.e. the maximum throughput of the USB connection between a mote and a WiFi router. During the first test, we sent 8000-bytes of application data payload per second (62.5Kbps)

over UART. This is equivalent to a CPS application collecting 2-byte samples from 4 sensors every 1ms. First, we established the limitations of two sensor platforms. The Z1 platform, operating on 16MHz, can send 80-bytes of sensor measurements every 10ms. The TelosB platform, operating on 8MHz, can send 96-bytes of sensor measurements every 12ms (the larger data size amortizes the serial packet header's overhead). Next, we measured the actual speed at which sensor samples can be collected. Using the faster Z1 platform, we observed a delay of 18-20ms in receiving the measurements from the Phidget motion and Phidget distance sensors attached to the mote through ADC; this is analogous to an application sending data at the rate of 1.735Kbps. We conclude that in those testbed configurations where every mote sends data over USB to the attached router, as shown in Figure 5.4(d), the bandwidth requirements for data collection are orders of magnitude lower than the Ethernet bandwidth. Therefore, the testbeds relying on the wired backbone network infrastructure do not utilize the Ethernet bandwidth resources. Next, we verify that the backbone network consisting of wireless infrastructure can also sustain the data flow generated by all motes transmitting over UART at the maximum rate.

One of the concerns of the framework is how well the backbone network operating on IEEE 802.11 ad-hoc mode can collect data from all sensor motes reporting simultaneously, especially in an indoor deployment where other WiFi networks are present. We tested the throughput of the WiFi mesh by downloading 1GB file from a server located right next to the network's gateway. On a single WiFi router, we observed download rates oscillating between 21.12Mbps and 20.73Mbps for indoor and outdoor deployments, respectively. When all routers were downloading at the same time, depending on each router's distance from the gateway, download rates ranged from 0.99Mbps to 2.17Mbps for the indoor deployment, and from 1.1Mbps to 4.451Mbps for the outdoor deployment.

During the experiments with all the routers downloading simultaneously, we observed one of the indoor deployed WiFi routers sporadically stalling downloads, whereas in the outdoor deployment 2 to 4 routers were always pausing downloading for few seconds. The network throughput variation on each router resulted from the dynamics in the network routing topology computed by the OLSR. In the indoor deployment, the routes were more stable, and only one out of 16 routers was more than one hop away from the gateway. Instead, in the outdoor deployment, 5 out of 14 routers were two hops away from the gateway. Despite the variations in the WiFi ad-hoc network routing topology, in both testbed deployments we observed that the wireless communication bandwidth was

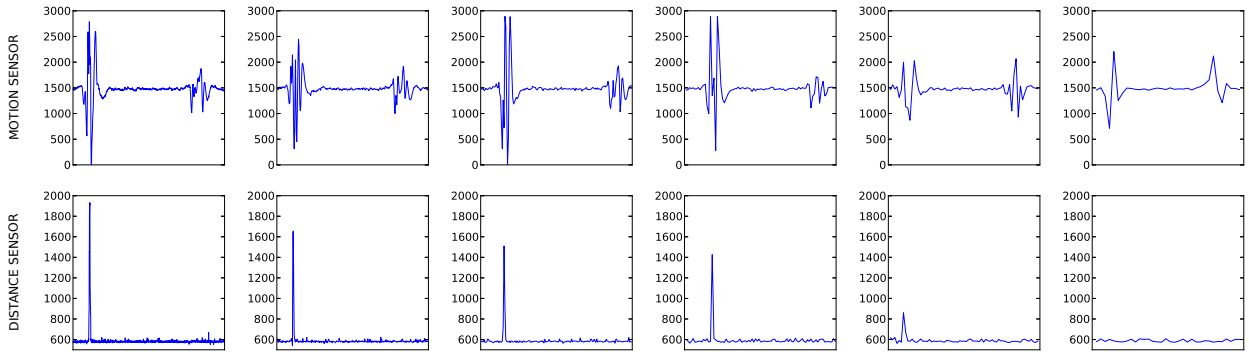
Sampling Delay (ms)	Packet TX Delay (ms)	App (bps)	Radio (bps)	Avg. Delivery (%)
15	300	2133	2906	87.94
17.5	350	1828	2491	92.41
20	400	1600	2180	97.36
22.5	450	1422	1937	99.30
25	500	1280	1744	99.76

Table 5.2: Average delivery of packets at the sink node.

orders of magnitude larger than the limits at which motes collect sensors' samples. The size of the presented testbeds does not allow us to evaluate the WiFi mesh network scalability of collecting the sensor data through a single gateway or to exactly estimate when more routers need to be connected to Ethernet to serve as gateways. We can confirm, however, that the presented examples of the network backbone resources are sufficient to collect sensor data. We conducted a 3-hour experiment with all motes sending data over USB at the maximum rate (62.5Kbps). During the experiment, all messages were successfully transmitted to the testbed server. Then, for over a year, both testbeds have been successfully collecting sensor data sampled at the rate of 10Hz.

These presented experiments show that the framework can support sensor sampling at the maximum rate at which the existing mote architecture can generate measurements, while relying solely on the WiFi-based backbone network, as shown in Figure 5.4(a). While high-frequency sensor sampling is helpful in understanding the testbed's surrounding environment, it is not practical for many CPS production deployments, which require both the IEEE 802.11 and the IEEE 802.15.4 wireless communication standard. Moreover, in many CPS deployments designers do not have the luxury of using WiFi at the final product version, i.e. WiFi installation may be too expensive or impractical to deploy due to power constraints.

Collecting Data Through LPWN. We continued the experiments and studied when CPS can collect sensor data through the LPWN infrastructure itself, as shown in Figure 5.4(b), instead of using the WiFi network. We compiled a firmware for Z1 and TelosB motes with an application simulating the collection of 2-byte sensor measurements from 2 sensors of each mote. Once 80-bytes of sensor samples are collected, the application running on each mote sends over the network



(a) 25ms (b) 100ms (c) 200ms (d) 400ms (e) 500ms (f) 1000ms

Figure 5.7: Motion and distance measurements from sensors detecting people walking through a doorway, for various sampling frequencies. In each experiment a person first walks through a doorway and then walks along the hallway next to the door. The motion sensor detects both events, while the distance sensor only detects a person walking through a doorway.

a packet with the sensors' measurements to one mote designated as the data sink. The sink mote operates as the LPWN's gateway to the WiFi network. The sensor data is routed by the Collection Tree Protocol (CTP) [62], running over the CSMA MAC protocol and radio following the IEEE 802.15.4 standard.

Table 5.2 reports the results of the experiments with 17 sensor motes collecting measurements through LPWN at one sink mote. Starting from the leftmost column the table reports: the rate at which each of the 2 sensors sampled measurements, the rate at which packets with 80-bytes of sensor data payload were sent over the network, the bandwidth generated by the application's data, and the bandwidth at which the radio sent packets - this includes sensor data payload together with the application, network protocol, and MAC protocol headers: a total of 109-bytes. The rightmost column of the table reports the network average delivery defined as a percentage of packets that were received at the sink mote. Each line of the table contains the average result of a separate one hour experiment.

We compared the results of the data collection experiments with the results reported in literature. CTP operating on CSMA MAC delivers 94.7-99.9% of the packages, depending on the testbed deployment [62]. In our experiments, we achieved above 97% of delivery when each of the

17 motes sent 109-byte long packets not faster than every 400ms. When the packet transmission delay increased, the average network delivery increased as well up to 99.75% for packets sent every 500ms. Next, we analyzed the network data throughput. For an application sampling sensors at the rate of 20, 22.5 and 25ms, the network sent data at the rate of 36.1, 32.1 and 28.9Kbps, with the delivery rate of 97.36, 99.30 and 99.76%, respectively. As a reference, the theoretical upper bound of the single-hop throughput for IEEE 802.15.4 is 225Kbps [155] (the standard defines bandwidth of 250Kbps). This physical limit is further impacted by the CSMA MAC protocol with unslotted random-access to the channel [1]. Further, the throughput decreases due to the overhead of the network and MAC protocols (periodic beacons, message acknowledgements, packet transmission back-off delays), motes' hardware limitations [155] and the dynamics in the wireless channel with links between the motes being bursty (shifting between good and poor quality) [23], [175].

We presented the application data collection and network throughput statistics showing how much sensor data can be collected through 17-mote LPWN, deployed within the framework in an indoor environment. These results provide a reference point for a user deploying the framework and collecting sensor data through the motes' wireless network instead of WiFi. While these results are sensor data agnostic, in the following examples we show traces of physical world measurements together with an analysis of how much sensor data is needed to detect physical events of interest.

Sensing for Event Detection. In the last set of the experiments we show examples of using the framework to understand how much sensor data has to be gathered to detect an event. Some events, such as change in temperature, do not require frequent sensor sampling. Thus collecting sensor measurements every one, three or even fifteen minutes is sufficient to detect such events. For other events, however, such as motion detection or occupancy estimation, the adequate sensor sampling frequency is not that straightforward to estimate. Next, we show tradeoffs between the number of taken sensor samples and the accuracy of the detected events.

On all the motes we deployed a firmware with an application detecting if a person walked through a doorway. In related work, motion and door sensors were used to detect occupancy in a home [138]. In another work, multiple distance sensors were used to track people walking between the rooms of a house [82]. In our indoor deployment we used two Phidget sensors attached to Zolertia Z1 motes: motion sensor and distance sensor, operating on 5V and 3V, respectively, and mounted on top of the door and facing downward.

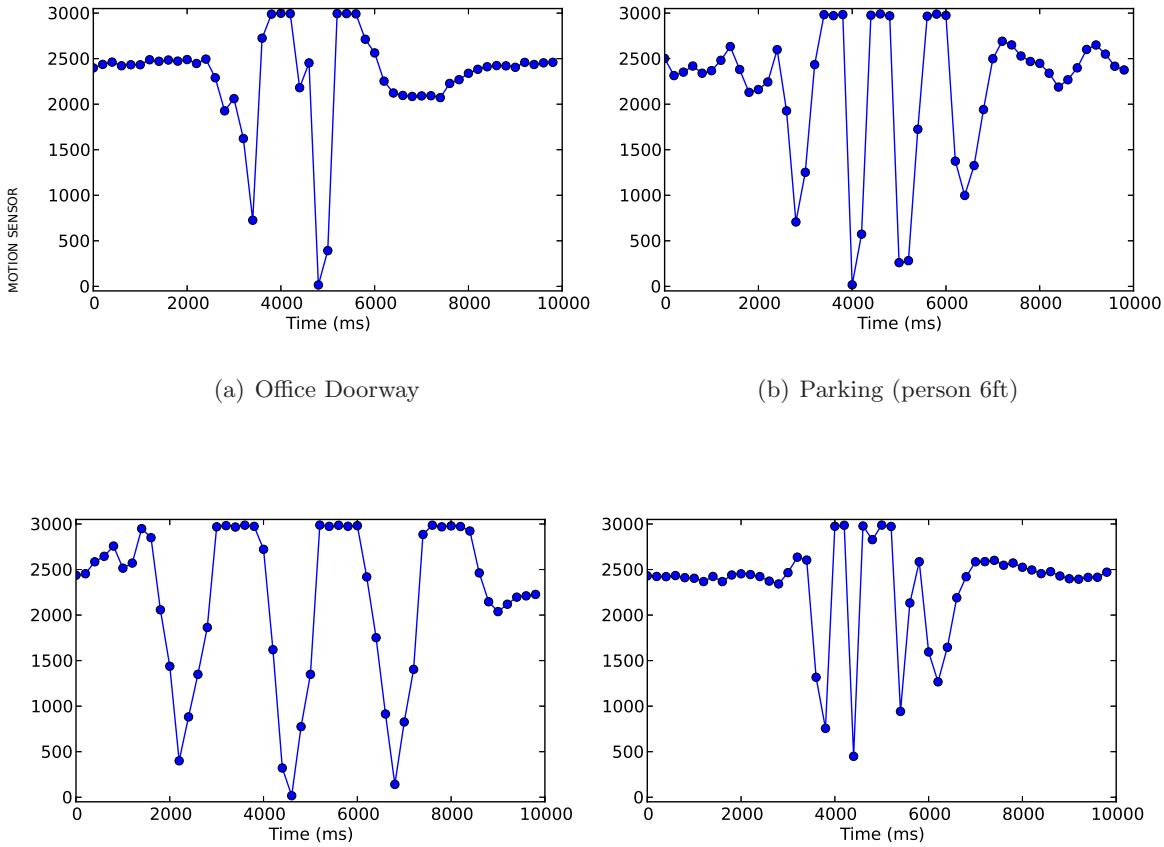
The goal of these experiments was to determine the frequency at which the two sensors should collect samples. In a work focusing on detecting the height of a person walking through a door, Hnat et al. observed that a head moving at a speed of 3 meters per second passes the sensing region of the distance sensor in about 100ms [82]. In our experiments, we started with sampling every 25ms because that is the highest rate at which 99.76% of packets are successfully delivered, as reported in Table 5.2. Then, we continued the experiments with longer sampling delays, studying the trade offs between the number of collected sensor data samples and the quality of the data for the event detection based on the visual observation.

Figure 5.7 shows sample results from six experiments with the motion and distance sensors detecting if a person walked through a door. For each experiment, the sensors' sampling frequency varies from 25ms up to 1000ms. During each experiment a person walked *through* a door and then, after a short delay, another person passed *by* the door. As shown on the upper graphs of the figure, the motion sensor detected people walking both through the door and by the door. For sampling rates of 25, 100, 200, and 400ms, the observed events could be positively classified between the two cases. When the motion sensor took samples every 500ms or longer, the measurements were not sufficient to distinguish if a person walked through the doorway or not.

The lower graphs of the Figure 5.7 show the measurements from the distance sensor. The distance sensor only detected people walking through the doorway, not people walking on the hallway. As the sampling frequency decreased, the amplitude value of the distance sensor raw measurement decreased as well, from 1973 to 1414 for 25 and 400ms delays, respectively. When sampling at the rate of 500ms or more, the distance measurements either did not indicate a walk through the doorway or, as shown in the figure, the sensing value was very low, often not distinguishable from the noise.

The indoor deployment case-study highlights the need for minimizing the impact of false-negative and false-positive events in CPS. In designing CPS, it is thus necessary to find the sampling rate that will lower the chance of miss-recognizing events. In some applications, like occupancy-estimation, it is crucial to use multiple sensor modalities to cross-validate the occurrence of events.

Understanding the Physical Phenomena. In the last experiment, we compare the motion sensor measurements from the indoor testbed deployment with the motion sensor measurements from the outdoor testbed deployment.



(a) Office Doorway

(b) Parking (person 6ft)

(c) Parking (person 12ft)

(d) Parking (car)

Figure 5.8: Motion sensor measurements samples every 200ms in indoor and outdoor deployments. The traces show sensor measurements when a person walks at a various distances from the sensor and when a car drives in front of the sensor.

Figure 5.8 shows traces sampled every 200ms for the same Phidget motion sensor (operating on 3V) detecting four events in the indoor and outdoor testbed deployments. Each chart shows 50 motion sensor measurements collected for a period of 10 seconds and with each single measurement marked as a dot. Figure 5.8(a) shows a trace of measurements from the motion sensor mounted on top of the door and recording when a person walked through the doorway. The remaining charts show traces of measurements gathered by the motion sensor installed 3 feet from the ground, on the parking's light pole. Figure 5.8(b) and Figure 5.8(c) show traces of measurements collected when a person walked in front of the sensor at the distance of 6 feet and 12 feet, respectively. Figure 5.8(d)

shows a trace of measurements taken when a car was passing in front of the motion sensor.

As shown in Figure 5.8, the values of the motion sensor measurements depend on the distance between the sensor and an object of interest, the speed at which the object moves, and the context of deployment. In the first three charts, we notice that people who walked in front of the motion sensor at a further distance spent more time in the sensing area, which resulted in longer event measurements with higher amplitudes. The last three charts compare different speeds at which objects moved in front of the sensor, indicating shorter event time and lower amplitude values for the car's motion detection than for peoples' motion detection, because cars move faster and consequently spend less time in front of the sensor. Finally, we compare the first chart from Figure 5.8(a) with the last one from Figure 5.8(d). The charts show similar measurements with events occurring for a similar period of time (approximately 3 seconds) in two different scenarios: a person walking through the doorway and a car driving on the parking lot.

The outdoor deployment case-study makes evident how critical the understanding of the context of the sensor deployment is to successfully detect and classify the event. Information such as the sensor's position, orientation and distance from objects of interest as well as the physical models of events need to be combined with the sensor's data. The meta-data describing the context of the deployment is as essential as the sensors' measurements themselves.

In conclusion, the experimental results presented in Figure 5.7 and Figure 5.8 confirm the importance of deploying CPS testbeds to understand the physical environment together with the behavior of the events of interest. Depending on the CPS application, the placement of sensors and their sampling frequency, the traces of events of interest have different characteristics and need to be studied at the beginning of the CPS development. High-frequency sensor sampling and measurement data collection are crucial not only for understanding the environment in which CPS is deployed but also for quantifying the quality of information retrieved from the sensors. Our tests show that early stage CPS prototyping and deployment is necessary to understand both the information impacting the control of CPS and the cyber technology tradeoffs, which influence the cost and the quality of CPS products. Therefore, the local testbed deployment process boosts both research and business in developing CPS applications.

5.6 Conclusion

We presented a new framework to assist engineers and researchers in the efficient deployment of heterogeneous wireless testbeds for CPS applications. Our framework addresses prevalent issues in multi-disciplinary CPS projects which rely on the actual deployment of control systems utilizing sensor and actuator peripherals connected together in a network of low-power wireless embedded devices. It provides software tools that simplify the setup of flexible testbed architectures for relatively low hardware costs. We presented the functionality of the framework on testbed deployments in outdoor and indoor environments, in industry and academia, respectively. The tools are shared through an open source project, thus allowing the research community to use the framework and encouraging contributions to its further development.

Chapter 6

Distributed System Services

This chapter describes and evaluates a distributed WSN run-time software framework and its programming language. The framework creates an abstraction of a unified and integrated system operating with a single, high-level program. The program specifies the execution of tasks on the distributed run-time infrastructure. The tasks implement either applications or system services, which provide abstractions for facilitating the WSN multitasking across the network. This work introduces two system services that allow tasks to communicate through a small memory shared across the system, and to synchronously schedule the execution of tasks across the network.

The abstraction of the single system running on top of the WSN and the programming language introduced in this chapter build on the concepts and works presented in the previous parts of this dissertation. From Chapter 3, this work adapts the notion of collectively scheduling the computation logic and the supporting communication protocols. Chapter 4 shows the need for system services using an example of the WSN run-time system self-monitoring. In this chapter, however, the system services provide abstractions that enhance the execution of multiple applications on the same WSN. The WSN testbed installation tools presented in Chapter 5 were used to conduct sensing and data collection experiments in academic and industrial environments. The experience from using these testbeds indicated the need for the system-level abstractions, which simplify the WSN programming and improve the run-time performance.

This work was done in collaboration with Omprakash Gnawali from the University of Houston and my advisor Luca P. Carloni.

6.1 Introduction

The WSN motes run with a firmware combining the application code together with a simple embedded operating system such as TinyOS, Contiki, Arduino or FreeRTOS. Usually, the firmware only includes application logic controlling the state of the hardware on which it runs. In WSNs, the program running on a mote also maintains the states of its communication protocols and its radio to enable ad-hoc collaboration with other motes. This distributed nature of the WSN increases the programming complexity and leads to challenges in attaining WSN scalability and robustness, especially when the independently running and loosely connected motes together collaborate on the same problem.

One alternative to the current WSN design methodology is to shift the programming perspective from the motes to the network, while at the same time hide the distributed nature of the system. The unreliable wireless communication over a network of resource constrained devices provides no system-level determinism, which in other computation technologies, such as PC or smart-phone, is the foundation of software development. Instead of programming the individual motes [43; 103], we propose to program the whole WSN as an integrated system.

In this chapter we present Chip and Dale that together create an abstraction of a system operating on top of the WSN motes. The system consists of a firmware compiled from a library of Chip system modules and a program written in the Dale programming language. Dale models the whole network behavior as a FSM, provides abstraction of network processes, processes scheduling, and supports communication between processes. Chip provides the services that enable to form the abstraction of a single system and to execute Dale program logic.

We start with a Chip and Dale tutorial. We introduce the system concepts and building blocks as well as the Dale programming language syntax and semantics by using simple code examples. Next we discuss the Chip underlying services. First, we present the *Best Effort Data Synchronization* (BEDS) protocol. This protocol provides communication between processes running on multiple motes in a WSN. BEDS allows Dale programmers to update a variable's value visible to all the motes in the WSN. Second, we present the *Estimate the End of the Dissemination* (EED) protocol. It introduces the notion of the WSN rendezvous. EED allows Dale programmers to assume that the same set of processes starts on all the motes at the same time.

The evaluation section of this chapter consists of two parts. First, we present a set of experiments

on the introduced system services. We study the limitations of the system services and trade-offs between the run-time overhead and the services' successful execution. Concretely, we look at how well BEDS disseminates data that allows network processes to communicate and what parameters impact BEDS performance. We run experiments against EED to understand the assumptions that a DALE programmer can make about starting processes synchronously on all the motes in WSN.

In the second section of the experiments we show an example of a low-power data collection application programmed in DALE. Our approach allows programmers to accomplish low-power operation by synchronously duty cycling the whole WSN, instead of turning on-and-off each mote's radio at random times. This technique lowers the per-mote duty cycle by ∼90% and saves ∼60% of battery power, when data is collected every 15 or 30 minutes.

In summary, the contributions of this chapter are the following:

- Introduce CHIP run-time system infrastructure and DALE programming language that together provide mechanism for executing multiple processes on the WSN.
- Introduce two protocols for synchronous process execution and shared data dissemination that provide system services abstractions used by application programmers to start processes and enable inter-process communication.
- Evaluate the performance of the system core functionality on two remote testbeds consisting of 30 and 100 motes.
- Demonstrate programmability and advantages of the presented system on a data collection application example.

6.2 CHIP and DALE

CHIP is a run-time system for WSN. It creates an abstraction of a single system, with computation and memory resources physically distributed on the loosely interconnected motes. DALE is a programming language for CHIP. With DALE a programmer specifies the set of processes that should execute across the network configured with CHIP. The processes include applications and system services supporting the execution of the applications on the unified system abstraction.

Related Work. CHIP and DALE are the second generation of the Fennec Fox and Swift

Fox, respectively. Presented in Chapter 3, the first generation of the run-time framework and the programming language demonstrated the complexity of executing two heterogeneous applications across the WSN. Because heterogeneous applications may have different performance requirements, which necessitates the use of different communication mechanisms, in Fennec Fox different applications run on different protocols. In Chapter 4, the computation and communication logic that is executing together on a network protocol stack is called a process. The process may have either application or system functionality. In this chapter we design a WSN system abstraction that runs heterogeneous processes executing applications and system services. The system services facilitate scheduling of processes and small data exchange among them.

Since the beginning of the WSN research, there have been efforts to organize individual motes into a form of a unified network architecture [52]. The prevalent approach aimed at a horizontal cut across the network protocol stack running on every mote [31], giving the rise to the IPv6 for low-power devices, known as 6LoWPAN [86]. In Chip, the layered stack architecture is also the foundation of deploying computation across the network; however, it is not bound to a single communication protocol, but at run-time it can synchronously change across the WSN.

To simplify WSN programming, researchers have proposed higher-level programming abstractions [28; 68; 109; 124; 140; 180]. The macroprogramming frameworks, such as sMapReduce [69], MacroLab [83], and Kairos [67], introduced new languages that permit to program the whole WSN as a single system. In these works, WSN communication is fixed to one protocol that does not perform well with all types of computations. The other limitation is that these programs execute only one application, making difficult to express multitasking on the WSN. Tasks have been introduced in Tenet [157], which however constraints the network architecture topology and does not allow the WSN to change communication protocols at run-time. In Dale, applications can be expressed as a single task executing across the network as well as a composition of tasks that can use the support of other tasks providing system-level services.

6.2.1 Concepts, Syntax, Semantics

A **process** consists of a computation and communication logic executing across the network. Each process has three **module** instances running concurrently on a protocol stack consisting of three layers: application, network, and radio, with one module per each layer of the stack. Modules im-

plement the process' computation logic or communication mechanisms that allow the computation to be executed across the network. Processes run module instances, *i.e.* the same module running in two different processes executes the same logic but with its own copy of the data.

Listing 6.1: Example defining process *collector*.

```
uint16_t destination = 0x64

process collector { temperature(60, destination)
                    ctp(destination)    cc2420(26, 0, 200) }
```

Listing 6.1 shows a snippet of a DALE program that defines a network process called *collector*. This process consists of three modules: temperature, ctp, and cc2420. These modules implement processing temperature sensor measurements, network routing with CTP [62] protocol, and wireless communication through cc2420 radio, respectively.

CHIP modules take **parameters** that regulate their execution. These parameters refer to reserved memory, which a module can read and write. Parameters are initialized with default constant values, unless specific values are set by the programmer. Listing 6.2 shows a code defining a module: its type, name, source code location, parameters, and the default parameters' values.

Listing 6.2: Definition of a module in DALE library.

```
use am cc2420 $(DALE_LIB)/am/cc2420 ( uint8_t channel=26,
    uint8_t power=0, uint16_t sleep=0)
```

The module parameters can also point to a shared memory created as a variable in the DALE program. DALE allows modules to communicate by sharing the same memory. DALE variables have two possible scopes. **Mote variables** have the scope of a single mote: changes in a variable value are visible only to the modules running on the same mote. **Network variables** have network scope and are synchronized across all the modules running on all the motes in the network.

In Listing 6.1, *destination* is a mote variable initiated to value *0x64* and passed as a parameter to the process' modules. A variable has a network scope when its name is preceded with **@** (at) sign. Network variables are disseminated by BEDS protocol further discussed in Subsection 6.2.3.

Daemons are processes that are constantly executing to provide CHIP system services, such as running the network scheduler or supporting interprocess communication. They are denoted in the DALE programming language by **!** (exclamation mark) following the process name.

Tasks are processes that must be scheduled to be executed. In CHIP, multiple tasks can run concurrently. The list of tasks running across the network is part of the CHIP state. The **state** is defined as a list of non-daemon processes that are running across the network. Thus, when the network changes its state, it switches the processes that are running on all the motes.

Listing 6.3 shows a code of a DALE program collecting sensor data. The program defines two processes, *sched* daemon and *collector* task. The task samples sensor data and sends the measurements to the destination. The destination address is specified as a mote variable *dest*. The WSN installed with this program starts running in the *sensing* state, which includes the *collector* task. The *sensing* state starts all the modules listed in the *sched* daemon and then modules listed in the *collector* task.

Listing 6.3: Defining state *sensing* running with *collector* task.

```
uint16_t dest = 10

process sched ! { StateSync() EED() cc2420() }
process collector { sample(1000, 0xFFFF, dest) ctp(dest)
                                              *cc2420(26, 0) }
state sensing { collector }

start sensing
```

Every process has separate module instances on the application and the network layers. Two processes running in the same state can instantiate the same module with different parameters. For example, let *ctp* be a network layer module implementing the CTP protocol and taking one parameter: the address of the destination (sink) mote. In CHIP one process may collect data with the *ctp* network protocol module having a different destination root mote from the destination of

another process running concurrently with the same module.

Multiple processes running in the same state may want to use the same radio with different parameter values. For example, different processes may want to run on a different radio channel. To determine how a radio should be configured, CHIP chooses the radio module parameters from the first process listed in a state, giving priority to processes that precede radio module name with * (star) in the DALE program. Therefore, at run-time only one radio module instance controls the radio at a given state.

After daemons and tasks, the third class of processes are events. **Events** are special processes that compute a Boolean function that can trigger the network to switch from one state to another. As processes, they may have their own communication mechanism, which can be used to define distributed events. For example, multiple smoke-sensors can calibrate their measurements before they set of a fire alarm. At a given state, only those event processes that can transition the network to a different state are executing together with the daemons and the current state's tasks.

CHIP **policies** specify event processes that can transition the network from running from one state to another. At run-time, the first policy that matches the current state and an event evaluating to *True* triggers a state transition. Each policy specifies the names of the states between which the transition can occur as well as the name of the event process that can trigger this policy and lead to switching to a new state. A mote enters a new state by starting event processes, then tasks and finally by resetting daemons.

Listing 6.4 defines two network states and two event processes called *after60Sec* and *fire*. When the network is in the *sensing* state, it runs all its tasks, all the daemons and the event *fire*. When *smoke* evaluates to *True*, CHIP finds the matching network state transition policy. In this example, when fire smoke is detected the network stops the current state's processes and switch to run in the *emergency* state. After running for sixty seconds in the *emergency* state, the network will stop the current state's processes and switch to run in the *sensing* state.

Each state has assigned a **priority** level. By default, each state has the lowest priority 0. A higher state priority is assigned in a DALE program by following the state's name with its priority level denoted by letter **L** and followed by a number. In Listing 6.4, the *emergency* state is defined with priority level 3, whereas the *sensing* state has priority 0.

Listing 6.4: Network state reconfiguration.

```
process sched ! { ...
process collector { ...

event after60Sec { timerSecondE(60) nullNet() nullAM() }
event fire { smoke() nullNet() nullAM() }

state sensing { collector }
state emergency L3 { ... }

from sensing goto emergency when fire
from emergency goto sensing when after60Sec

start sensing
```

By default, the state transitions and interprocess communications affect only the motes on which they occur. The network reconfiguration feature as well as the synchronization of the network variables are not enabled, unless explicitly specified. For example, when an event process triggers a state transition and starts running new processes, only the mote detecting the event changes its state, not the rest of the network. Similarly, when a module writes to a network variable, this memory change is visible only to the modules on that single mote. Thus, by default network variables act like mote variables. This approach lets DALE programmers to have a better control over the CHIP firmware, and let them consciously add or avoid services that consume motes' memory and computation resources.

CHIP provides daemons for synchronizing the state and the network variables. The first daemon process called *sched* ensures that all the motes in the network are running in the same state, with the same tasks and event processes. The *sched* process runs with an algorithm that disseminates state updates across the network and resolves network state inconsistency by either randomly synchronizing the network to one of the states or deterministically converging the network into the state with the highest priority level [182].

Another CHIP daemon called *cached* synchronizes the values of the network variables on all the motes in the network. When a process' module running on a mote writes a new value to a variable with the network scope, CHIP updates this variable on all the motes in the network and notifies the modules using this variable about the content update.

6.2.2 Network Data Dissemination - BEDS

DALE variables, and particularly the variables with the network scope, have been introduced to provide communication between multiple processes running on the same WSN. While the problem of data dissemination in WSN has been already addressed from multiple perspectives, the previous works did not guarantee zero beacon overhead when the data stays consistent and when no new motes join the network. This is particularly important in duty cycling WSNs, where repeating beacon transmissions is waisting scarce energy resources. Another limitation of some data dissemination techniques is the difficulty of sharing across the network data that consists of multiple variables, which need to be independently updated.

Related Work on Data Dissemination in WSN. Popular data dissemination techniques focus on minimizing the overhead coming from notifying and maintaining data updates. Trickle [128] is commonly used to keep track of changes in the data by disseminating short beacon messages with the data summary. In DRIP, the summary is a simple version number [192] and DIP disseminates hashes of version numbers of multiple data items that it keeps track of [133]. In CodeDrip [38], network coding is further used to efficiently merge the updates. One of the limitations of the previous works is relying on the data summary to differentiate between data versions, instead of using the data values themselves. With these approaches it is possible that the same variable is changed into two different values while having the same summary representation, for example the same version number.

CHIP takes the best-effort, domain expert approach to synchronize network variables, allowing data semantics to be used when inconsistency is detected. Programmers of the CHIP modules can include logic that deterministically resolves variables' values inconsistencies by understanding variables' semantic, particularly by knowing what the variables represent and how they are used by the modules. This technique applies to multiple variables, with different semantics and shared among multiple CHIP processes.

Data Hash and Zero Beacon Overhead. CHIP network data dissemination aims for zero beacon overhead when the data stays unchanged. Instead of only tracking data summaries CHIP provides the infrastructure for maintaining data synchronization on all the WSN motes, relying on the variable's version number, its content, as well as variable's semantics. Writing the same data into a variable does not cause any network overhead, whereas simultaneous update of two different

variables is detected. To notice the changes in the shared data, CHIP sends the CRC hash of all the network variables that have the network scope, instead of sending the data version summaries.

CHIP avoids sending beacons with data summary by piggybacking on every radio packet transmission. The lower bits of the data CRC are stored in the header of every radio packet. By piggybacking on all the network packets, CHIP avoids sending extra Trickle beacons and detects inconsistency in the data only when the motes are communicating and not sleeping.

The Best Effort Data Synchronization (BEDS). CHIP variables with the network scope are used to share a small amount of data across the WSN. The data is exchanged among the motes by a many-to-many (M2M) network protocol called BEDS. The size of the shared data must fit into a single radio packet. At compile time, DALE computes the size of all the variables with the network scope together with their associated meta-data, which includes each variable's 1B sequence number. Then, DALE checks if the total size is less than 100B, which is the maximum amount of data stored on every mote and synchronized across the network with other motes.

We find that when a larger amount of data must be shared or transmitted across the network, it is better to run a dedicated CHIP process with a network protocol optimized for a given data exchange. For example, to collect sensor data one should use a process with CTP [62] instead of relying on the CHIP network variables. Similarly, a dedicated process should be used to disseminate various control parameters to each actuator.

The BEDS protocol provides the best-effort data synchronization. Every mote has a copy of all the network variables called ***vLocal***. Each variable implementation consists of a fixed memory storing the variable value and a version number associated with the variable. When a variable value is updated, its version number is set to the largest value from all variables' versions, plus 1. A variable's value and its version are accessed by ***value()*** and ***version()*** functions respectively.

The BEDS protocol broadcasts copies of *vLocal*. A new BEDS message is sent (1) after a CHIP module changes a value of a network variable, (2) CHIP detects data inconsistency based on the CRC piggybacked on the overheard or received packets and (3) when the protocol receives a BEDS message, called ***vReceive***, that differs from *vLocal*.

Algorithm 3 shows the BEDS protocol actions executed on a mote after receiving a BEDS message from the network. If the received *vReceive* is equal to *vLocal* then BEDS exists. Otherwise, BEDS starts comparing pairs of variables L and R from *vLocal* and *vReceive*, respectively. For every

Algorithm 3 Best Effort Data Synchronization

```
1: Input: vLocal and vReceive
2: if vLocal = vReceive then
3:    EXIT
4: end if
5: for L ∈ vLocal and R ∈ vReceive do
6:    if value(L) != value(R) then
7:       if version(L) = version(R) then
8:          version(L) ← version(L) + RANDOM
9:          value(L) ← value(R)
10:         newValue(R, conflict = TRUE)
11:      end if
12:      if version(L) < version(R) then
13:         version(L) ← version(R)
14:         value(L) ← value(R)
15:         newValue(R, conflict = FALSE)
16:      end if
17:   end if
18: end for
19: BROADCAST
```

received variable R with a value different from L and a higher version number (line 12), the local variable copy L updates its value and version number with those of the received R variable. Then, BEDS notifies Chip about the new value R by calling *newVariable*(R) function. BEDS detects data inconsistency when the local L and the received R variables have different values, but the same version number (line 6-7). Then, the variable version number is increased by a random positive number and the value of L is updated with R. Next, BEDS indicates the detected data inconsistency by calling the *newVariable* function and passing the new value R and setting *conflict* parameter to TRUE. In the end, BEDS broadcasts the updated *vLocal*.

The BEDS approach to resolving data inconsistency in the network has two benefits. First, when a mote receives a variable with a different version number, the received value is not lost but

signaled to CHIP modules using this variable. When CHIP receives *newVariable* function call, it finds all the modules that use this variable and sends them notification with the value update, passing information about the potential conflict detected by BEDS. CHIP modules may disagree with the new variable values and correct them with new data. Overwriting a variable with a new value increases this variable version number and forces the BEDS protocol to broadcast a new copy of *vLocal*. Second, by randomly increasing a variable's version number, BEDS attempts to synchronize the variable across the network. This solves a problem occurring when CHIP modules do not act to resolve the version conflict and motes continue to update back-and-forth the same variable with different values.

In summary, BEDS provides a mechanism for updating a set of variables across the network. BEDS synchronizes the variables by tracking their version numbers. By reporting variables with inconsistent values, BEDS leaves deterministic synchronization of the data up to the CHIP modules that understand variables' semantics.

6.2.3 Synchronized State Reconfiguration - EED

A DALE state transition creates a network rendezvous, a point in time when the CHIP stops running some processes on all the motes in the WSN and starts others. The network rendezvous is an important abstraction of the system functionality because it allows programmers to assume that the processes start simultaneously on all the motes in the network. To allow DALE programmers to rely on this abstraction, the network rendezvous must be very precise. This requires dissemination of a message with a request to change the WSN state and execute switching between the states almost instantaneously on all the motes in the network. However, a naive dissemination of a message requesting state transition can lead to different motes in the same network starting the same process with a random delay measured in seconds. Such random delay is too high for applications often running with millisecond (ms) timer precision [1] and periodically executing for only few hundreds of ms.

Related Work on Time Synchronization in WSN. Protocols, such as TPSN [60], FTSP [142], PulseSync [121], VHT [167] and Glossy [55], narrowed down the network time-synchronization error with respect to one root mote, from tens of microseconds (μs) down to sub-μs scale. The

[1] Commercial FreeRTOS applications use periodic *tick* to measure time in ms, with *tick* period often set to $10ms$.

improving time accuracy techniques require better understanding and more deterministic control of the mote architecture, especially radio devices, and event software interrupts execution [55]. In industrial deployments, consisting of heterogeneous architectures and running with different radio manufacturers these methods are difficult to use. The challenges are coming from inconsistent interpretations of the same code by different microcontroller compilers, porting the same low-level code into different architectures, and making assumptions regarding the low-level radio operations including the MAC protocols as well as the time it takes the software to process radio packets. Thus, it is difficult to establish robust distributed system abstractions by applying methods that make any assumptions about deterministic radio transmissions in the low-power mesh networks.

In WSNs, the μs time-scale synchronization precision is useful for TDMA-like MAC protocols [20; 203] and applications estimating a location of an event, such as a sniper firing a bullet [174]. In many other applications, such time synchronization precision is not necessary either because applications operate on the ms scale or because they can tolerate larger time synchronization errors. One such example is the acoustic-sensor-based event detection, where a network synchronized down to $\sim 3ms$, gives $\sim$1 meter single mote distance estimation error, which is further mitigated by multilateration of measurements from multiple sensors. Thus, the precision of time synchronization is dictated by the applications' requirements, which in the WSNs, often can be sufficiently fulfilled by other techniques for establishing a point of reference in time. Such an alternative approach is to use some form of rapid flooding [139; 55; 36] to denote an occurrence of an event in the network.

Estimate the End of Dissemination (EED). CHIP runs a scheduler process that disseminates information requesting to switch the state across the motes. The scheduler request is communicated over the Estimate the End of the Dissemination (EED) network protocol. EED disseminates data and establishes network rendezvous. Instead of explicitly synchronizing the motes' time with respect to some point of reference, such as a designated mote in the network, the rendezvous is achieved by synchronously finishing dissemination on all the motes at the same time. The maximum pairwise time difference between any two motes finishing dissemination is the EED error, which is on the order of ms.

EED disseminates the application data payload together with the EED header that contains necessary information for establishing the network rendezvous. The EED header includes two

32kHz time values. The first time value represents the length of the dissemination period. The second time value represents the estimated amount of local time that is left on the sending mote till the end of the dissemination. During the EED dissemination period, the motes countdown the estimated time that is left till the end of the period. As motes exchange messages, they compare their estimates of how much time is left till the end of the dissemination. Whenever a mote receives an estimate that is smaller than its own, it sets its estimate to the received one. Thus, the motes try to minimize the amount of time that is left till the end of the dissemination. With this approach, the first mote starting EED ends the dissemination after the time equal to the EED period length. The rest of the motes in the network will try to estimate this end and finish disseminating at the same time. The end of the EED dissemination is the rendezvous that serves for the WSN system and application processes as a global point of reference.

The length of the dissemination period impacts the EED precision. A longer period improves dissemination reliability together with the accuracy of the estimation of the end of the period. A shorter period reduces the time and energy used during packets exchanges. The EED period can be changed between different state transitions. This allows WSN designers to vary the reliability and robustness of the EED dissemination. For example, when the network transitions to an emergency state, where it is critical for every mote in the network to be alarmed, the EED period can be increased to improve the dissemination reliability.

EED does not require the selection of a single root mote as a point of reference: the protocol works when a single mote or multiple motes start data dissemination with this protocol. While similar techniques of global time synchronization require from tens-of-seconds to tens-of-minutes to achieve network-wide μs synchronization [142; 121], EED spends few hundreds of ms to achieve network rendezvous with the error on the order of ms. EED has no assumptions about the MAC protocol nor the radio drivers, except requiring the radio to timestamp messages.

With EED, Dale programmers can create time-based events triggering WSN state transition and can expect processes to start on all the motes almost simultaneously. Figure 6.1 shows how EED allows Dale programmers to schedule timers that fire on all the motes in WSN. In the figure, from left to right, the network runs in the state S_1 until event e_1 occurs. This event can be observed by one mote or many, simultaneously or with various delays. The first mote that notices the event, checks if event e_1 triggers a transition from state S_1 to S_2, and when it does, it starts the EED

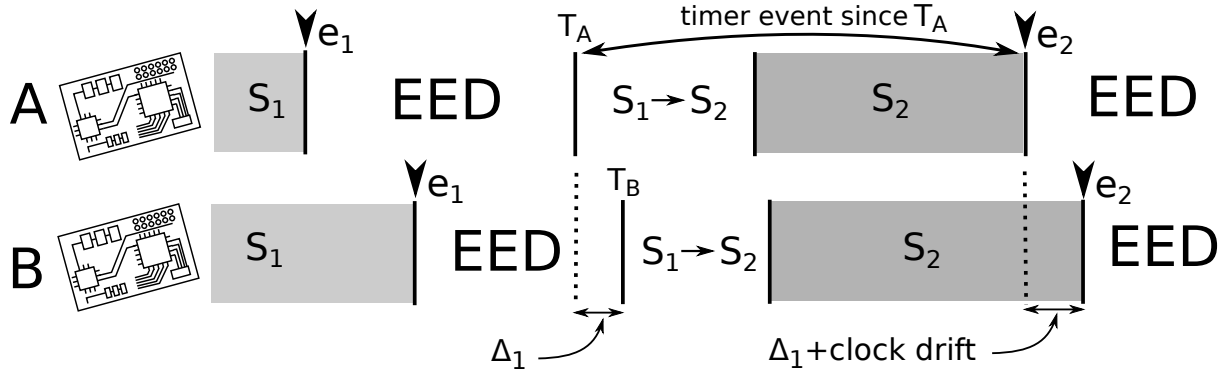


Figure 6.1: Synchronized Network Processes with EED.

dissemination requesting all the motes to switch to S_2. Then, mote B receives a message from mote A, which includes the time T_A denoting how much time is left till the end of the EED dissemination at the mote A. When T_A is lower than T_B or T_B is not yet set, mote B updates its estimate of the rendezvous. Otherwise B drops the message.

By retransmitting the EED messages and adjusting their estimates of the end of the dissemination, all the motes continue to minimize the network rendezvous error. Figure 6.1 denotes the rendezvous error as Δ_1, measuring the time between the first and the last mote that finishes dissemination of the EED messages. At the end of the EED dissemination, the protocol saves the actual local time of the rendezvous occurrence in its module parameter variable.

The local time-stamp of the rendezvous can be shared with all the Chip modules running on the mote through a variable that has a mote scope. Thus, the rendezvous time-stamp can be used by other modules as a point of reference. In Figure 6.1, e_2 is a time-based event that should fire on all the motes simultaneously. When this event is started with respect to the last time when the EED dissemination ended, the motes' timers fire within the error of the past rendezvous (Δ_1) skewed by the local clock drift on the motes. This requires the EED period to be long enough to allow the motes to resynchronize, taking into account their potentially increasing deviation from their past estimate of the rendezvous.

6.3 Experiments

We run experiments on two remote testbeds: FlockLab [132] and Indriya [35]. These testbeds are instrumented with the same mote platform TelosB [163], which has 8MHz Texas Instruments MSP430 microcontroller, 10 KB of RAM, 48 KB of ROM and a 2.4 GHz IEEE 802.15.4 radio. Located in Zurich, FlockLab has 30 motes and provides very precise time-stamping of GPIO traces and power measurements. Indriya has 100-motes deployed indoor across a three-floor building of the University of Singapore. Indriya has only serial interface for sending logging messages, which impacts the precision of time-stamping the logs. Also, data sent over the serial is often delayed by at least few random milliseconds.

6.3.1 CHIP Services Evaluation

We start with evaluation of the presented Chip services: the dissemination and synchronization of the network variables and the accuracy of the network rendezvous after state transition. First, we run stress tests against the BEDS protocol to find its limitations in data synchronization. Second, we run tests against the EED protocol to understand what impacts the maximum-pairwise error between two motes in the network at the end of the EED dissemination. We present results that evaluate the trade-offs in achieving different performances of those protocols and which serve as a point of reference for developing applications.

BEDS Evaluation. In the first set of experiments run on Indriya, we measure the accuracy and the delay of synchronizing Dale global variables using BEDS. In our experiments we focus on the BEDS performance metrics that are important to WSN application developer using the Dale variables which have network scope. Therefore, our experiments show how fast the network variables are updated on all the motes, and how often an update with a new value fails to reach all the motes in WSN.

We prepared a test application that sets a new value to a Dale network variable on every mote. The test application called *TestDataSync*, receives five variables passed as parameters to its module's instance. At random delay, with mean specified as another module parameter, the application picks one of the variables and sets it to a random value in a range [0, 65535]. During the experiments, we changed one of the 5 variables in a randomly chosen mote every random delay,

with a mean set to $250ms$, $500ms$, 1, 2.5, and 5 seconds.

The BEDS algorithm is implemented as an application module. BEDS runs on top of the *rebroadcast* network protocol, which is one of the network-layer modules implemented for CHIP. When *rebroadcast* receives a message with an application's payload, it broadcast the message and repeats transmission for zero or more times. This network protocol is configured with the number of retransmissions of the same payload and the delay between consecutive retransmissions. The protocol automatically increases the delay when it detects a network congestion. In our experiments, the *rebroadcast* retransmission delay is set to $10ms$. We try a different number of retransmissions chosen from the set {0, 1, 3, 5, 15}. All experiments are run twice: once with motes having their radios always turned on and once with their radios duty cycling every $200ms$.

Listing 6.5: DALE program testing BEDS protocol.

```
# Program: BEDS_stress_test.sfp

uint16_t @v1 = 0
uint16_t @v2 = 0
uint16_t @v3 = 0
uint16_t @v4 = 0
uint16_t @v5 = 0

process cached ! {  BEDS()
   rebroadcast(1, 10) # repeat, repeat delay
   cc2420(26, 0, 0) } # channel, power, duty cycling

process testBEDS { TestDataSync(250, v1, v2, v3, v4, v5)
                                    nullNet()    nullAM() }
state testing { testBEDS }

start testing
```

Listing 6.5 shows the complete source code of the DALE testing program. First, we define five variables with the network scope. Second, we create *cached* daemon with the *BEDS* application and *rebroadcast* network protocol. Next, we create the *testBEDS* task, which includes the *TestDataSync* application. This task runs in the *testing* state, which is the initial state in which CHIP starts running.

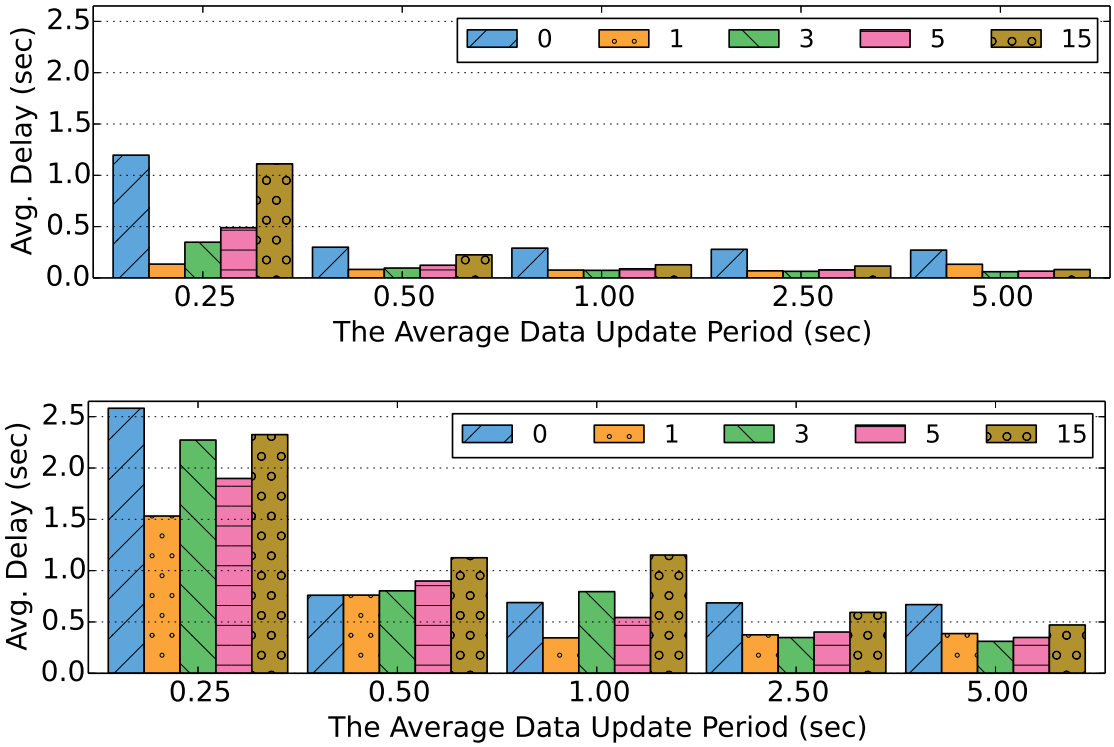


Figure 6.2: The average BEDS dissemination delay measured in all experiments for variable update with a random period. The periods are chosen from different uniform distributions of given means. Experiments repeated for different numbers of the BEDS payload retransmission. On the top, experiments with radio always running; on the bottom, experiments with radio duty cycling.

BEDS Average Delay of Updating a Variable in WSN. Figures 6.2-6.3 show results from the experiments measuring the dissemination delay of a network variable with BEDS. On the graphs, the x-axis shows the average period at which *TestDataSync* application sets a new value to one of the network variables. The y-axis shows the average delay it takes the BEDS protocol to update a variable with the new value on all the motes in the network. Figure 6.2 shows all experimental results and Figure 6.3 shows results from 99% of the experiments with the shortest delay. In each figure, the graph on the top shows the results with radio always turned on and the graph on the bottom shows the results with radio duty cycling. These results indicate that three main factors impact the average dissemination delay.

First, the time to update a variable on all the motes in the network depends on how often the variables values are changed. When variables values are modified too frequently, the dissemination

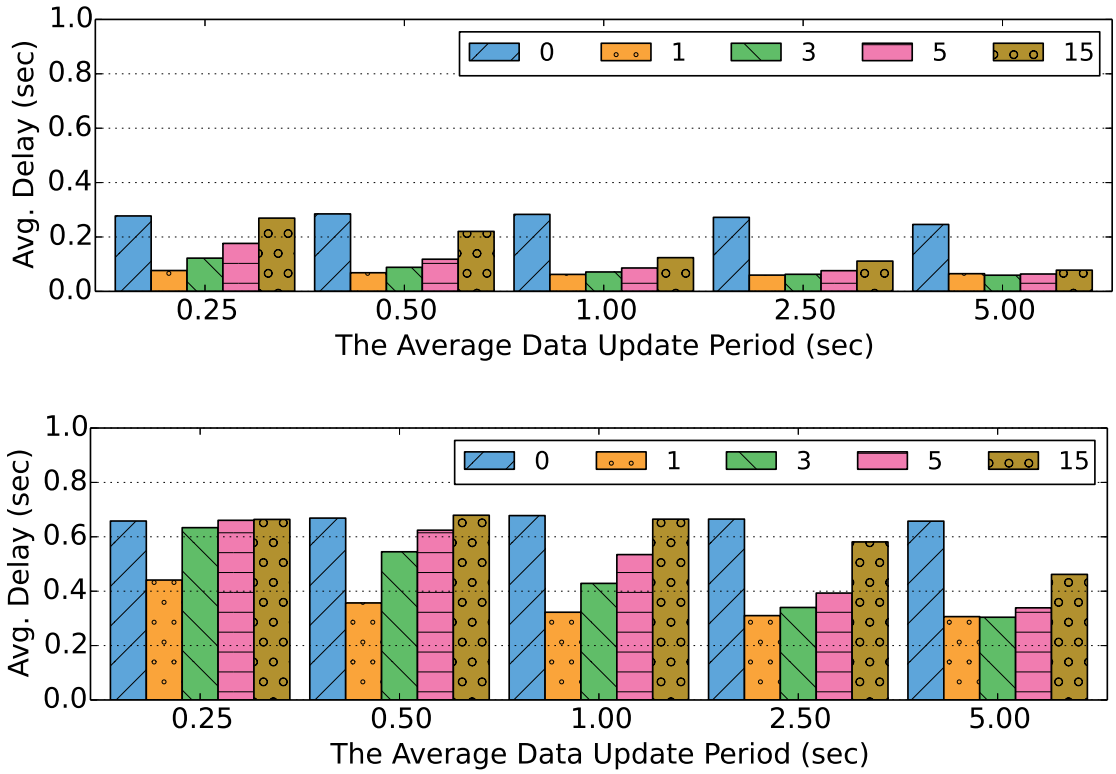


Figure 6.3: The average BEDS dissemination delay from the best 99% of the experiments for variable update with a random period. The periods are chosen from different uniform distributions of given means. Experiments repeated for different numbers of the BEDS payload retransmission. On the top, experiments with radio always running; on the bottom, experiments with radio duty cycling.

delay increases. In the particular Indriya WSN testbed, the variable dissemination delay increases significantly when the values are changed every $250ms$ and the radio is always running. When the radio is duty cycling, changing variables every $500ms$ or less increases the dissemination delay. The longer delay is a result of the network congestion: when variables are updated frequently, the MAC protocol has to back-off its broadcast due to the detection of other ongoing transmissions.

Second, the configuration of the radio on which BEDS is running impacts the dissemination delay. A network with radio duty cycling requires more time to update a variable on all the motes then the case when the radio is always turned on. In most experiments, duty cycling radio triples the dissemination delay.

Third, the dissemination delay depends on the number of repeated BEDS message transmissions by the *rebroadcast* network protocol. Transmitting a BEDS message only once (the blue bars) and without any following retransmission usually results in the longest delay since it increases the chance of a packet being lost or collided with other transmissions. Retransmitting a BEDS message too many times, *e.g.* 15 (the brown bars) increases network congestion. In the experiments run on this particular testbed, retransmitting once or three times results in the lowest delay. In 6 out of 10 experiments retransmitting once results in the shortest delay, and this is the default configuration that we use in our applications.

The results from Figure 6.2 indicate that when BEDS messages are retransmitted only once, the average dissemination delay stays below $200ms$, with radio always on. When radio is duty cycling and variables are changed every $250ms$, the delay is $1.5sec$. The delay is $750ms$ when variables are updated with the mean period of $500ms$, and the delay is less than $410ms$, with updates sent every $1sec$ of longer. However, as shown in Figure 6.3, in the best 99% of the experiments the average delay is much lower. When radio is turned on, the variables are disseminated in less than $85ms$. The dissemination takes $370ms$ when radio is duty cycling and variables are updated every second or more.

The 1% of the longest delays comes from the experiments when there was a conflict, resulting from two motes changing the same variable into two different values within a very short period of time. Because the testing application is not participating in correcting the data inconsistency, the BEDS protocol has to resolve the collision by randomly increasing a conflicting variable's sequence number. In few rare cases, up to 5 iterations of randomly increasing the sequence number were necessary to establish consistency among all the motes.

Comparing both figures, the probability of data conflict increases when the average variable update period is equal or less than $0.5sec$. Conflicts are also more likely to occur when the same message is retransmitted too many times, particularly 15; this gives more time for two or more random motes to modify the same variable. These conditions further increase network congestion which intensifies the severity of data inconsistency.

In our experience with runnings different applications than those presented here the variables are changed on the order of seconds or more and BEDS very rarely needs to solve data conflict, but when it does, it ensures to resolve it. Further applying data semantics in most cases guarantees

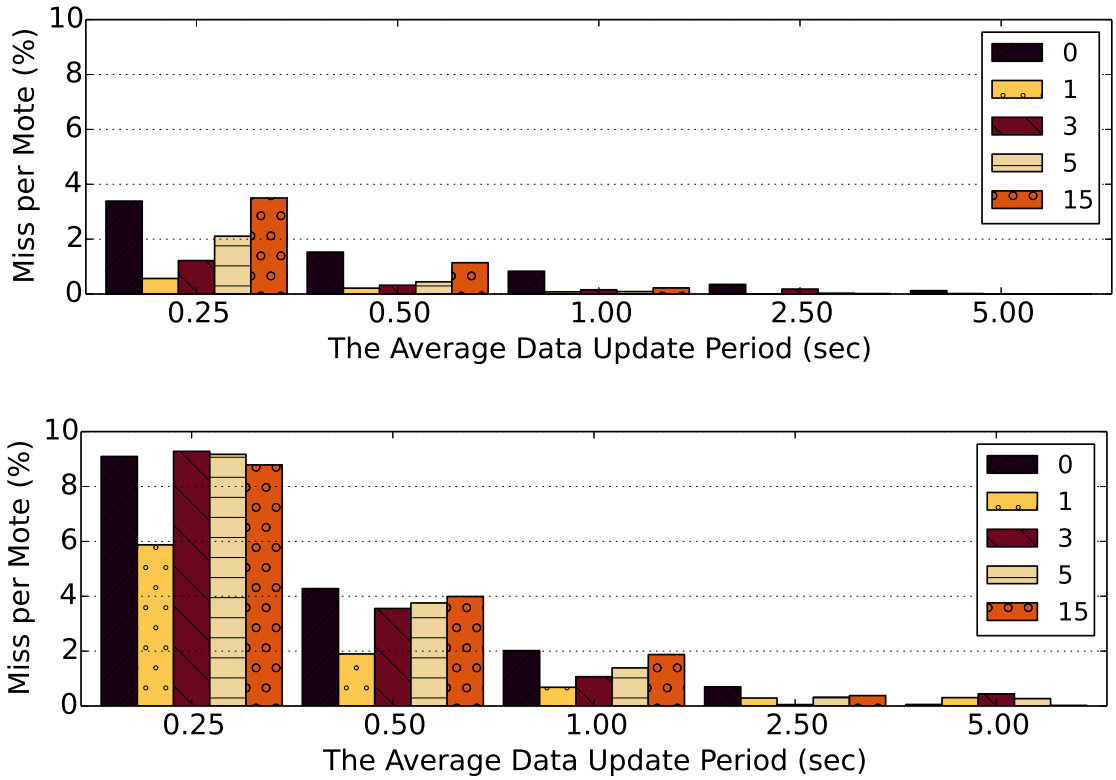


Figure 6.4: Percentage of motes that are expected to not receive data update for variable update periods with different means. Experiments repeated for different number of the BEDS payload retransmissions. On the top, experiments with radio always running; on the bottom, experiments with radio duty cycling.

at most one extra dissemination iteration to establish a variable's value consistency. For example, when a variable stores a maximum value, the modules that use this variable can solve the value's ambiguity.

BEDS Chances of Failing Updating a Variable. Figures 6.4-6.5 show results from the experiments measuring probability of the BEDS network protocol to fail dissemination of a new variable's value across the network. On the graphs, the x-axis shows the average period at which *TestDataSync* application sets a new value to one of the network variables. On Figures 6.4, the y-axis shows the percentage of motes that are not updated with the latest variable value. On Figure 6.5, the y-axis shows the probability at which BEDS fails to disseminate a new value to all the motes in the network. In each figure, the graph on the top shows the results with radio always

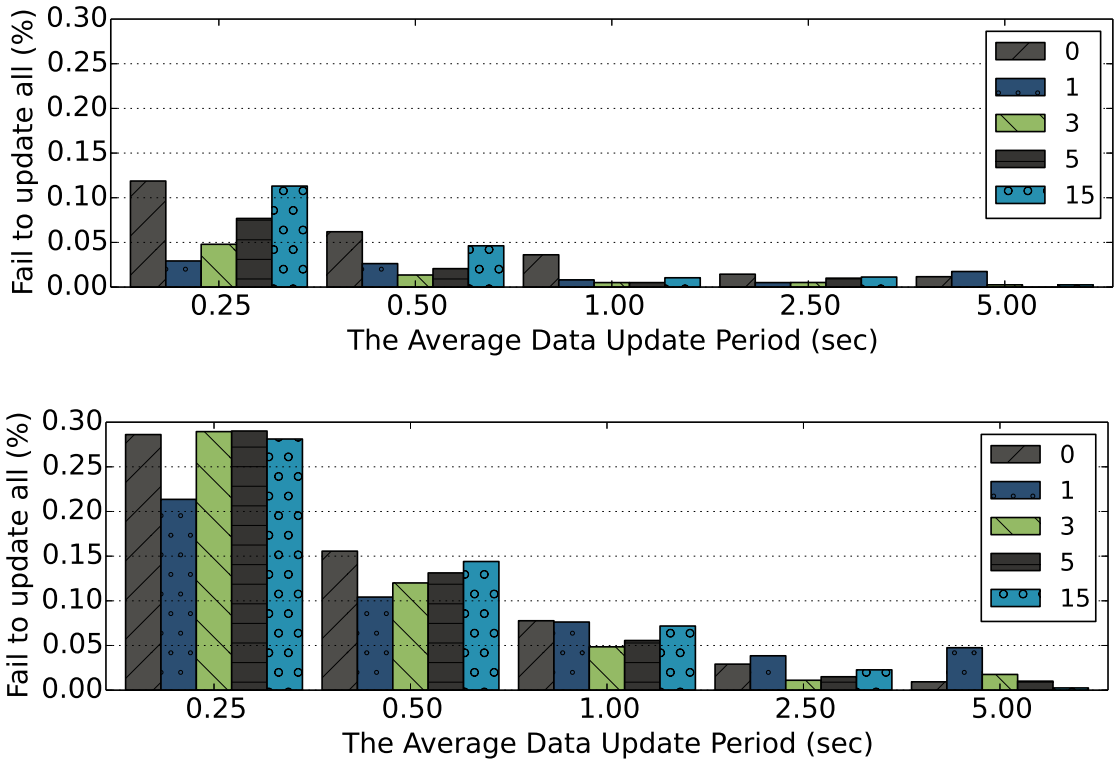


Figure 6.5: Chance that a new value update will not reach every mote in the network for variable update periods with different means. Experiments repeated for different number of the BEDS payload retransmissions. On the top, experiments with radio always running; on the bottom, experiments with radio duty cycling.

turned on and the graph on the bottom shows the results with radio duty cycling.

The same factors that impact BEDS dissemination delay also influence the chance of failing to disseminate update to all the motes in the network. Duty cycling the radio at least doubles the chance of missing an update across the network. Configuring *rebroadcast* without or with too many retransmissions has also a negative result on successful data dissemination. In Figures 6.4-6.5, half of the experiments with the smallest number of BEDS failures in updating data on all the motes comes from *rebroadcast* configured to retransmit the application's payload only once. Finally, updating variables at the average period equal or less than $500ms$ increases the chance of failing to reach all the motes.

The results from Figure 6.4 show the likelihood that a single mote in the network misses the

data update. In the experiments with *rebroadcast* resending BEDS message only once, the chance of a mote missing an update is 0.57% and 0.21% when variables are changed with the mean period of $250ms$ and $500ms$, respectively, for radio always turned on. When data is updated every second or more, the chance of an update miss at a mote is less than 0.08%. With radio duty cycling, the chance of loss drops from 5.87% to 1.9% and less than 0.7%, when the average data update period is set to $250ms$, $500ms$ and longer, respectively.

In some situations, for example when we update a sensor sampling rate on all the motes, we want to know the chance of a new data dissemination failure, which is the chance of one or more motes in the WSN not receiving an update with a new value. These results are shown in Figure 6.5. With *rebroadcast* set to a single retransmission, less than 0.03% of new values do not reach all the motes in WSN, when variables are updated with the mean period set to $250ms$ and $500ms$ and radio always turned on. With radio duty cycling, 0.21% and 0.1% of updates miss at least one mote when variables are changed every $250ms$ and $500ms$, respectively. When variables are updated with a period equal or longer than $1sec$, 99.992% and 99.319% of variables are successfully updated across the WSN, with the radio always turned on and duty cycling, respectively.

Next, we analyze the data logs from Figure 6.4 and Figure 6.5 to better understand the data dissemination failures in experiments with *rebroadcast* resending the BEDS data one more time after the initial broadcast. The average number of variable updates missed per mote, shown in Figure 6.4, is mostly dominated by those updates that missed all the motes in the network: this is the reason why the average chance of missing an update on a mote is higher than the chance of missing one or more motes per WSN.

In experiments when the values are updated with a period shorter than a second, 7.4% of updates failed to reach any other mote than the one that changed a variable's value. These dissemination failures resulted from other motes overwriting the same variable with a different value at the same time. In such conflicts, BEDS priorities the incoming data and gives application modules the opportunity of overwriting the variable with a historical or a different value, by applying the variable's semantic. Within the experiments where dissemination fails to reach every mote, in 54% of them only one mote missed a variable update, and in 66% of those experiments, out of 100 motes, five or less were not successfully updated.

EED Evaluation. In the next set of experiments run on Indriya and FlockLab we measure

the accuracy and the overhead of establishing network rendezvous through state transition with the EED protocol. We identify two main factors that impact EED performance: (1) the size and the scale of the WSN deployment, and (2) the WSN radio configuration that either keeps the radio turned on or is duty cycling. With respect to these two considerations, the clock drift on the motes has secondary impact on the EED error. The goal of our experiments is to find the EED limitations that can help application developers to better formulate assumptions about the underlying, network-wide CHIP services. Therefore, our experiments show how long it takes to switch the network state with EED, the overhead of the EED execution measured in terms of radio active time and energy consumption, and the precision of the EED synchronization measured as the maximum global time difference between two motes switching to a new state.

EED Synchronization Precision. In the first set of experiments we measure the EED accuracy. We use DALE test programs to trigger a state transition every minute either by a single mote or all of them. On both testbeds we measure the EED error on the ms scale. This error is denoted in Figure 6.1 as Δ_1. The EED period is set to $600ms$, and the EED is restarted at a random delay period, with mean set to 1 minute.

We report the median, average, and the maximum EED error between any two motes in the WSN testbeds from all experiments trials. On FlockLab, the average EED error is $0.6ms$, median $0.6ms$ and the maximum is $1.0ms$, measured by tracing the TelosB mote GPIO that indicates the end of the EED dissemination. On Indriya, the average EED error is $35.5ms$, median $33.8ms$ and the maximum is $53.9ms$, measured by time-stamping the serial messages sent by TelosB to the testbed at the moment when the EED dissemination finished. Unfortunately, this way of measuring the EED error has a limitation: the serial messages sent by the mote are randomly delayed when the microcontrollers in the motes are busy processing those messages that continue to be received by the radio at the time when serial message transmission is scheduled.

We repeated the EED testing experiment on the Indriya with a change in the code such that there are only two motes placed in the opposite corners of the testbed that report serial message exactly 10 seconds after the rendezvous. This change effectively lowers the overhead of sending serial messages on the mote side and collecting serial data on the testbed side. In this test, we measured $6ms$, $7.2ms$ and $18ms$ for the median, average, and the maximum EED error between those two motes. In summary, on the Indriya testbed we observe some larger error due to the specification

of the deployment environment, in particular, the structure of the building and the difficulty in propagating messages across the building's floors. However, we also acknowledge the limitations of testing time-synchronization on testbeds that do not provide precise tracking of events, the feature that FlockLab was designed for.

State Transition with EED in Radio Duty Cycling. In the next experiment we measure the EED performance on the WSN with motes' radio duty cycling, with the low-power listening (LPL) [151; 162] period set to either $200ms$ or $500ms$. In the experiments we try different EED dissemination period lengths, to find those that ensure successful state transition. With radio duty cycling every $200ms$ and EED period set to $900ms$, 99.3% of the motes are successfully reconfigured on Flocklab, synchronizing the network below $1ms$ error. With radio duty cycling every $500ms$, the EED period is set to at least $1700ms$ to achieve similar results. On Indriya, the minimum EED periods are at least $1600ms$ and $2400ms$ for radio duty cycling every $200ms$ and $500ms$, respectively, to reach 98.9% of motes. The EED error is between $64ms$ and $213ms$. The longer EED periods and larger EED errors often result from mistakes in the radio time-stamping of the received packages. In duty cycling network EED more likely disseminates messages in burst, which increases the chance of missing incoming packet time-stamp. Without correctly receiving a time-stamp, EED approximates the time when the packet was received, thus adding an error into the estimate of how much time is left till the end of the EED dissemination period.

EED Rendezvous for WSN-Wide Duty Cycle. Chip rendezvous serves as a network-wide tick. This tick can be used to synchronously put the whole WSN in a deep-sleep, with radio turned off for much longer periods than with traditional LPL approaches. To achieve synchronized deep duty cycle among the motes, EED accurately estimates the end of the dissemination period and stays on for enough time to compensate the motes' clocks drifts.

In the following experiment we run a Dale program with one state operating the radio with 0.00992% duty-cycle (waking up for $6.5ms$ every 64 seconds). Every 30 minutes a time-event triggers self-transition to synchronize the network and to provide new rendezvous point-of-reference for another 30 minutes of the network-wide deep sleep period. In this experiment, motes can only resynchronize when all of them wake-up together at the same time.

Listing 6.6 shows the complete source code of the Dale program testing EED with WSN-wide duty cycling. EED runs as a network protocol in the *sched* daemon process. The Chip state

transition adapts the Fennec Fox reconfiguration protocol [182], presented in Section 3.3, and runs it as *StateSync* application on top of EED. We create a *sync_time* event that fires every 30 minutes from a point of reference passed as a time-stamp parameter. In this program, the *sync_time* point of reference refers to *emarker* variable that has a mote scope and is shared with the EED module. At the end of the EED dissemination, the module updates the *emarker* with the time-stamp of the last rendezvous. In the program, the network runs with *idle* state that does not have any tasks. With the event-controlled period, the network triggers self-transition from *idle* back to *idle*. This implements a network-wide tick, which periodically resets the *idle* state on all the motes.

Listing 6.6: DALE program testing the EED protocol.

```
# Program: EED_test.sfp

uint16_t esrc = 0xFFFF    # Mote(s) starting EED
                          # 0xFFFF indicates all motes
uint32_t emarker = 0      # Last rendezvous time-stamp

process sched ! { StateSync()
         EED(600, emarker)    cc2420( 26,   0,   65534) }
                             # channel, power, duty cycle

event sync_time { timerMinuteE(30, esrc, emarker)
                                    nullNet()   nullAM() }
state idle { }

from idle goto idle when sync_time

start idle
```

We conducted experiments on both testbeds with the EED testing DALE program, varying the length of the EED period from 200ms up to 700ms. We measured the average WSN duty cycle, specifically the amount of time the radio is turned on while running the DALE program. This total active time represents the overhead of running a WSN with the network-wide deep-sleep duty cycle provided by CHIP.

Table 6.1 reports the experimental results, with average, maximum, and median values of the EED error. Each row, shows results for a different length of the EED period. In all experiments, all

EED	FlockLab				Indriya			
period	ave	max	med	%	ave	max	med	%
200	209	213	202	0.038	145	254	141	0.044
300	4.9	7.0	5.4	0.041	186	388	179	0.057
400	3.0	3.8	3.8	0.049	107	205	90	0.058
500	1.8	4.2	0.66	0.068	42	142	67	0.067
600	0.69	0.84	0.61	0.057	59	173	80	0.068
700	0.59	1.0	0.37	0.057	64	121	68	0.069

Table 6.1: The EED error measured in ms at the WSN rendezvous, for different EED periods lengths reported in ms.

motes successfully receive EED message that trigger the state transition. On the smaller FlockLab testbed, the average error is on order of few milliseconds for an EED period of at least $300ms$. On Indriya, the error stays below $100ms$ in most experiments with the EED period of $400ms$ or more. By default, we run EED with the period set to $600ms$, since it provides below $1ms$ synchronization in almost all experiments run on FlockLab.

The network-wide duty cycle with Chip EED protocol has very low overhead. The similar results on both testbeds report less than 0.07% duty cycle. The results are similar because in both testbeds the network sleeps for the same amount of time, using the same LPL protocol. In an ideal scenario, without considering software delays and false-positive busy-channel detections [153; 170; 212], the overhead of LPL itself is 0.0102%. Inspecting the logs we observed random false-positives in the LPL protocol, each one costing $100ms$ of radio stay-on time.

6.3.2 Application Examples

The second part of the experimental section demonstrates the Chip services in assisting programming and execution of low-rate data collection application. We start with a summary of the related work and then introduce the problem with the current design methodology. Next, we present a Dale-based solution, and run experiments on both testbeds. Our results demonstrate how Chip and Dale can be used to express the application logic and introduce features that make the WSN

simpler to install and more efficient to operate.

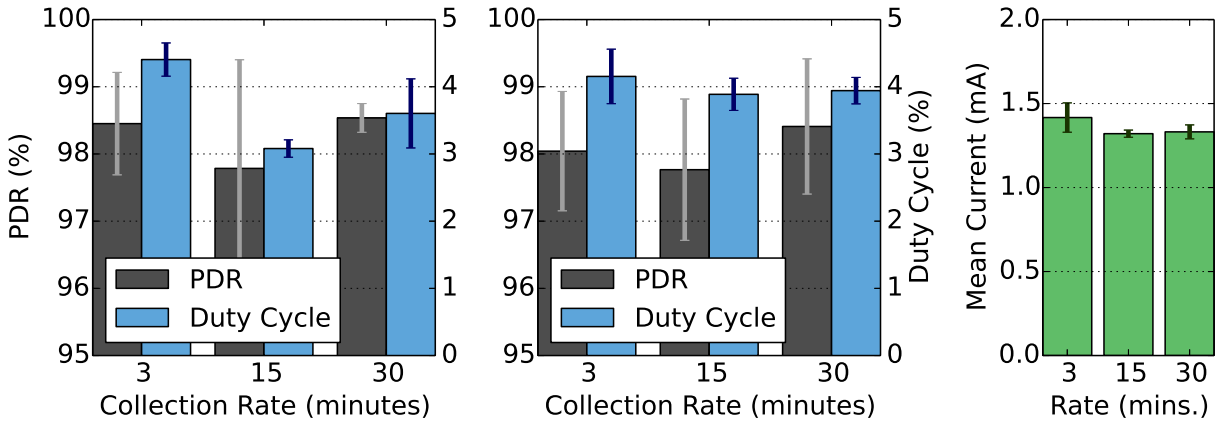
Related Work on Data Collection. CTP [62], BFC [164], ORW [113], and ORPL [44] are network protocols that collect sensor measurements while duty cycling the radio between 0.48% and 4.6%. Other works [170; 212; 40] ensure low-power data collection by solving the problem of radio false-positive detections while duty cycling [153; 170; 212]. There are also dedicated systems that can further enhance power efficiency [20; 153], reporting duty cycle between 0.15% [27] and 0.41% [54].

Data Collection Limitation. The presented network communication protocols for data collection are mostly application agnostic. A programmer can write an application collecting various amount of sensors' data and send messages every 30 seconds or less. However, we often find commercial installations with a data collection period on the order of minutes or longer. This is due to the costs of sampling sensors, which often constraints the amount of data being collected from a WSN that is expected to operate for years. Further, in home and in commercial building installations, application designers are looking to trade-off the data collection rate for longer lifetime, especially during the time when homes and buildings are not occupied. Unfortunately, with the current WSN application design methodology, we cannot save much of the WSN run-time power consumption by lowering the data collection rate.

In the prevalent WSN design approach, the data collection rate has relatively small impact on the WSN power consumption due to radio operation. Motes operate with a fixed duty cycling period that consumes power even when the application is not collecting data. From the application perspective design, when WSN is doing nothing, it should minimize the power consumption. In the mote-oriented programming paradigm there is no straightforward approach to address this problem in a matter that would support any data collection application, yet took advantage of the known sampling period.

We empirically verify the severity of this problem. As a benchmark, we use the TinyOS default data collection application with the CTP network protocol and BoXMAC together with Low Power Listening default sleep interval of $200ms$. On both testbeds we set a corner testbed mote as the data destination sink. In our tests, every 3, 15 and 30 minutes, each mote sends 100 bytes of payload toward the sink.

Figure 6.6 shows results from experiments collecting data at different rates. The grey (dark)



(a) Indriya (b) FlockLab (c) FlockLab

Figure 6.6: The WSN average operation power vs. the application data collection rate.

bars report the average and the standard deviation of packet delivery rate (PDR), which is the percentage of packets that were delivered to the sink. The blue (light) bars show the WSN duty cycle. The experiments from FlockLab also present the average current draw from the motes measured in mA.

With over 97% PDR, the radio continues to operate for over 3% of the time in all data delivery rates. On Indriya, the radio duty cycle is between 4.405% and 3.603% when data collection period is at 3 and 30 minutes. For the same data reporting periods measured on FlockLab, the radio duty cycle is 4.154% and 3.942%. In terms of power consumption, this 0.21% lower duty cycle on FlockLab saves $0.086mA$, from $1.417mA$ down to 1.331 mA. These results show that decreasing data collection rate ten-fold does not imply the same scale of lowering the power consumption.

Data Collection with Chip an' Dale. Our approach to data collection builds on the notion of state introduced in Subsection 6.2.1. With the network-oriented design, we distinguish two main states: the state executing application and the idle state when the application is not running. The current mote-oriented application design methodology does not support the notion of WSN-state, thus such application programming approach is not straightforward.

Figure 6.7 shows the FSM model of Chip based system running with data collection application. The model consists of three states: *snooping*, *collecting* and *sleeping*. When WSN is in the *collecting* state, the radio stays turned on and all the motes report their data to the collection sink. When

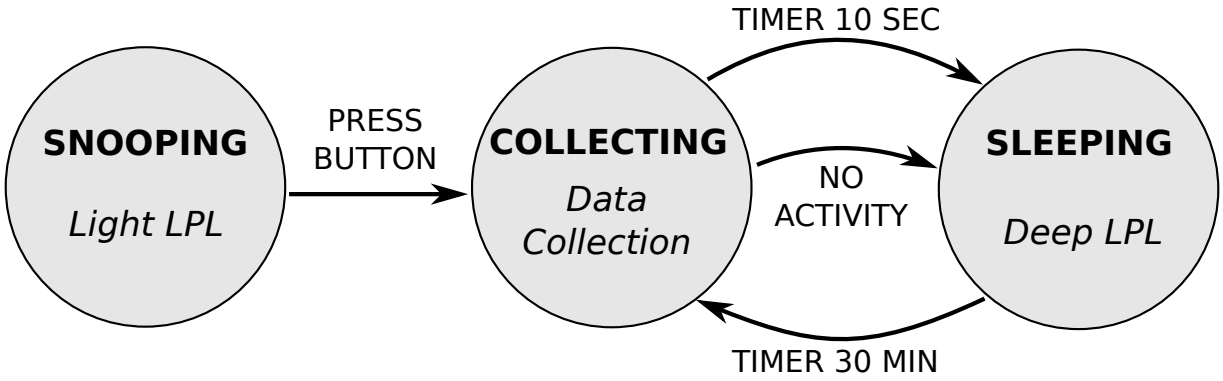


Figure 6.7: FSM model of network duty cycle.

WSN is in the *sleeping* state, the radio operates in a deep-sleep mode, turning on for $6.5ms$ every 64 seconds. The third state, *snooping*, runs with the default, $200ms$ duty cycle. When turned on, all motes start in the *snooping* state and wait either for a button to be pressed on one of them, or for the EED dissemination of a message requesting to switch to *collecting*. The purpose of the *snooping* state is to enable installation and further addition of more motes into WSN when it is already operational.

Once installed, the network switches between *collecting* and *snooping* states, as shown on Figure 6.7. Every 30 minutes an event triggers all the motes to wake-up and transition to the *collecting* state. Motes switch back to the *sleeping* state either after 10 seconds, or when the mote collecting the data, the sink, observes no further data collection activity.

Listing 6.7 shows the complete source code of a DALE program implementing the FSM logic presented in Figure 6.7. The network starts running in the *snooping* state. This state has no tasks and only one transition event called *installed*. The network will change its state when the button is pressed on any mote. A mote joining the network will synchronize its state during the next EED dissemination, which will have a different state id and a higher state's sequence number [182]. At the end of the EED dissemination, the EED network protocol saves its last rendezvous time-stamp in *emarker* variable that has mote scope. This variable is further passed to the time-based events as a point of reference to start measuring time.

The sensor data is sent to sink by *data_collection* task. This task runs in the *collecting* state. There are two events running in this state: *time_to_sleep* and *completed*. The first event fires at the sink 10 seconds after it is started. The second event, *completed*, monitors the application traffic

at the destination mote and fires when there is no more data to be received. This event is driven by the *noActivity* application module, which through the *tracker* variable monitors data collection traffic at *sample* application to learn how many motes reported the data in the past and how much data is received since the last data collection began. In the shown configuration, *completed* will not fire at the first time, since it needs to learn how many motes are in the network, but it will fire every consecutive time when for $200ms$ there is no application activity and 95% of packets are already delivered.

When not collecting, the WSN is saving energy by running in the *sleeping* state. This state does not include any tasks. It sets the radio with the *system_control* daemon configuration, that wakes it up for a brief time every 64 seconds. While sleeping, all motes are running with *time_to_work* event. This event counts how many minutes passed by since the last rendezvous time-stamp stored in the *emarker* variable. The event fires after this count matches the number of minutes stored in the network variable *period*, which by default is set to 30. Then, the motes switch back to *collecting* state and synchronize again from their last rendezvous errors and their local clocks drifts.

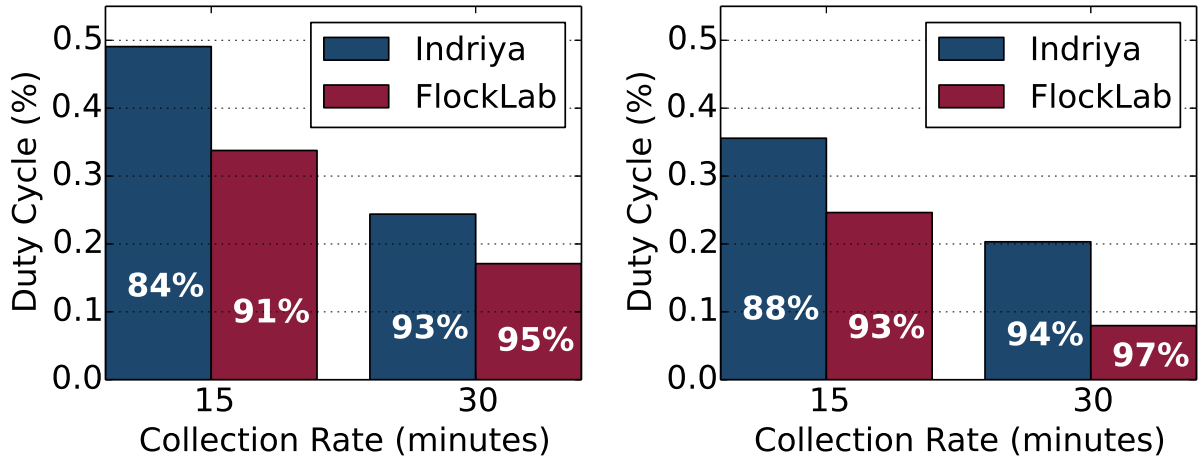
The application sampling period is controlled from outside of the WSN, through *serialVar32* application. This application runs as a daemon process that allows a WSN administrator to set a new sampling period. However, since the network operates in a deep-sleep mode for most of the time, BEDS can only synchronize this variable during the EED dissemination and when the CTP network protocol routes the application data toward the sink. This means that the new sampling period rate is effectively set during the next data collection.

We test the low-power data collection with Chip on both testbeds. In the program, we adjust the *dest* address mote for a sink placed in the corner of a given testbed. At run-time, we set the *time_to_work* event to fire after 15 and 30 minutes. In our experiments, all motes successfully switch between the states. On both testbeds we measure the average duty cycle and compare it against the default WSN data collection implementation running LPL with $200ms$ interval.

Figure 6.8 shows the results from experiments collecting data every 15 and 30 minutes. In all presented results, the application collected over 97% of packets (PDR $>$ 97%). Figure 6.8(a) reports the average duty cycle on both testbeds with the default Chip configuration of the EED set with $600ms$ dissemination period and a conservative configuration of the data collection, delaying application transmission by 1 second since rendezvous. This delay allows CTP to perform network-

Listing 6.7: DALE program with network-wide duty cycle.

```
# Program: Chip Low-Power Data Collection

uint16_t dest = 10          # Address of the sink mote
uint16_t src = 0xFFFF       # Address of all the motes
uint16_t tracker = 0        # Tracks application activity
uint32_t emarker = 0        # The last EED rendezvous
uint32_t @period = 30       # Data collection period

process system_control ! { StateSync()
                EED(600, emarker) cc2420(26, 0, 65534) }

process cached ! { BEDS() rebroadcast(1, 10) cc2420()  }

process vars ! { serialVar32(period) nullNet() nullAM()}

process booting { nullApp() nullNet() *cc2420(26, 0, 0)}

process data_sampling { sample(1000, src, dest, tracker)
                          ctp(dest)  *cc2420(26, 0, 0)}

event installed { button(1, src) nullNet() nullAM()    }

event time_to_sleep { timerSecondE(10, dest, emarker)
                                    nullNet()  nullAM() }

event time_to_work { timerMinuteE(period, src, emarker)
                                    nullNet()  nullAM() }

event completed { noActivityE(200, dest, 0.95, tracker)
                                    nullNet()  nullAM() }

state snooping { booting }
state collecting { data_sampling }
state sleeping { }

from snooping goto collecting when installed
from collecting goto sleeping when time_to_sleep
from collecting goto sleeping when completed
from sleeping goto collecting when time_to_work

start snooping
```



(a) Default

(b) Best

Figure 6.8: Data collection savings with WSN duty cycle.

ing adjustments. We set the detection of the end of the collection activity for 200ms. These are CHIP "off-the-shelf" results, which do not take into account the size of the WSN installation nor require any system tuning.

Figure 6.8(b) reports the best performance results produced by manually tailoring the application and CHIP parameters to the specific testbed installation. These results show potential for further improvement when the WSN can be reprogrammed with different versions to attain more optimal performance. However, these optimal system configurations per testbed so far can be only found by trial-and-error.

Figure 6.8 shows the results from experiments with the WSN running with the same data collection application, CTP network protocol, and the same LPL MAC protocol as they were used in the results from Figure 6.6, but this time configured with the CHIP WSN-wide duty cycle. The results indicate significant energy savings of the WSN motes, while performing the same application with the same successful data collection outcome but with the network operation configured differently at the DALE program level. On Indriya, the default CHIP configuration runs with 0.49% and 0.24% duty cycle, for applications collecting data every 15 and 30 minutes. With respect to the default TinyOS program, these are 84% and 93% lower duty cycle periods. On FlockLab default configurations run with 0.34% and 0.17% duty cycle, down from the benchmark 4% duty cycle,

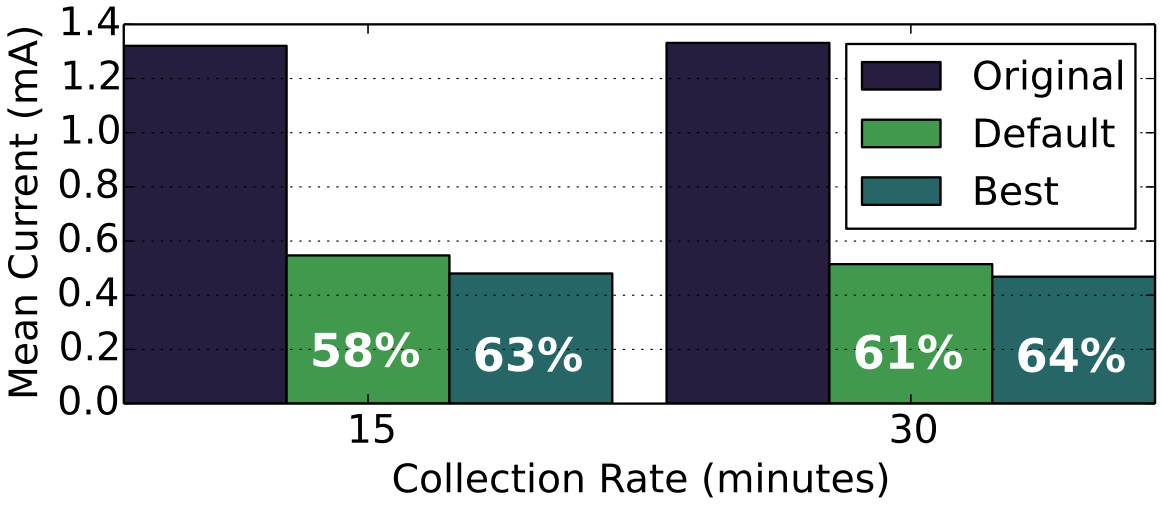


Figure 6.9: Average current draw and the resulting power saving in data collection application running on FlockLab and configured with Chip and Dale.

which is 91% and 95% lower, respectively.

Tuning the system parameters, especially shortening the length of the EED period and decreasing the delay before starting data collection adds more duty cycle savings. In all tries, there is at least 1% of additional improvement. In the best configuration on FlockLab, collecting data every 30 minutes results in lowering duty cycle by 97%, to the point that radio is turned on for only 0.079% of the time.

Figure 6.9 shows the average current draw per mote while running Chip with application collecting data every 15 and 30 minutes. The figure shows three current draw measurements. For comparison with the benchmark performance, the leftmost (dark-blue) bars represent the original current draw consumption measurements from experiments shown in Figure 6.6, which run with the TinyOS default per-mote LPL set to 200ms sleep period. The middle (light-green) bars represent the current draw consumption measurements with the same data collection application, but with WSN running with Chip in its default configuration. These current draw measurements relate to the FlockLab duty cycle results from Figure 6.8(a). The rightmost (dark-green) bars represent current draw consumption measurements with WSN running with Chip parameters optimized for the FlockLab testbed. These current draw measurements relate to the FlockLab duty cycle results from Figure 6.8(b).

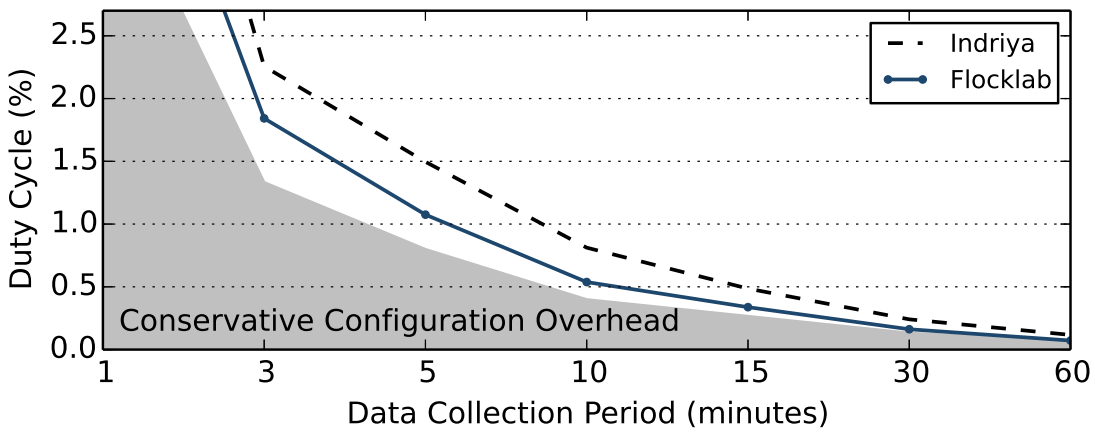


Figure 6.10: An average duty cycle on a mote running with CHIP and an application collecting data with different periods.

Running the same application as well as the same network and MAC protocols, but with CHIP and DALE saves two-thirds of power on TelosB motes. In the best CHIP configuration for FlockLab, over 99.5% of the application messages are delivered to the sink every 15 and 30 minutes. At these data collection rates, the motes consume on average $0.48mA$ and $0.468mA$, respectively. These current draw rates save 63% and 64% of power comparing to the benchmark run-time current draw consumption of $1.311mA$, with has the same application data collection rates.

Figure 6.10 shows the results from experiments with the network-wide duty cycle configured with different periods that match different application data collection delays. In these experiments, CHIP and the EED protocol were set to the default configuration. The x-axis shows the period at which the WSN was duty cycling and collecting data. The y-axis shows the average radio duty cycle across WSN. The gray area shows the percentage of time when the radio is turned on and the network is not collecting the data. This area represents the CHIP run-time configuration overhead. This overhead comes from: (1) running EED twice for $600ms$, once to switch to *collecting* state and once to switch to *sleeping* state, (2) the data sampling delay set to $1sec$ and (3) detecting the end of the data collection period in *noActivity* event $200ms$ after the last packet. Thus, the total run-time overhead, which does not depend on the length of collecting data by CTP, is $2.4sec$ in the default, conservative CHIP configuration.

Figure 6.10 shows results from both testbeds: Indriya (dashed-line) and FlockLab (continues

blue line). The figure shows that even for shorter collection periods, Chip achieves competitive power consumption results with respect to the traditional per-mote duty cycle approaches. At the 3 minute period, the WSN duty cycle for 2.256% on Indriya and 1.842% on FlockLab. The lower duty cycle percentage on FlockLab results from faster data collection in the smaller size testbed. For 5 and 10 minute periods, the average duty cycle is 1.496% and 0.812% on Indriya, and 1.075% and 0.538% on FlockLab. For one hour data collection interval, the default Chip configuration gives 0.117% of duty cycling on Indriya, and 0.072% on FlockLab, which translates to $0.496mA$.

For a short period data collection, it is better to run Chip in a single state, with the default protocol stack configuration running constantly LPL. Switching the states every 1 minute results in 6.672% and 5.299% duty cycle on Indriya and FlockLab, respectively. Collecting data with the same period and without network-wide sleep state results in duty cycle lower by $\sim$2%.

In summary, Chip an' Dale allow programmers to control the whole WSN as a single system, introducing the notion of the network state, which can be used to differentiate between the state when the network runs with an application and the state when the network is idle and should save energy. This WSN programming approach shows improved energy consumption results in data collection application, despite running on top of the network and MAC protocols that have been shown to be less effective in the low-power operation than the more recent ones [40; 113; 44; 164]. Therefore, there is a chance that by replacing the data collection application's network and MAC protocols with those newer ones, Chip an' Dale can achieve even better results.

6.4 Conclusions

We study the problem of the WSN programmability in the context of running multiple processes on the same hardware installation. Our solution consists of a programming language with semantic expressing execution of multiple processes on a single WSN. The programs run on all the network devices configured with a firmware that provides abstraction of a single, integrated system. Besides executing the applications, the motes also run with system services, which provide system abstractions that simplify the WSN programming. Our experimental evaluation demonstrates the programmability and the advantages of our WSN abstractions. The reported results show the performance of the introduced protocols used to implement the WSN system services and, using

application examples, demonstrate the WSN run-time performance improvements.

Chapter 7

Conclusions

In this last chapter, I outline the key milestones of my research contributions. I summarize the industrial and academic projects that adapted the software that was developed as part of this research. On the last pages, I describe promising directions for the future work.

7.1 Contributions

The primary contributions of this dissertation are outlined below:

- Defining a notion of the WSN state that represents a set of tasks executing across the WSN, with each task integrating computation logic with communication protocols.
  - Implement and evaluate a single WSN operating with two heterogeneous applications: a data collection application and an emergency application responding to an event detected by the network itself;
  - Establish that using the same MAC or network protocol is insufficient to obtain acceptable performance across a set of applications that require different types of communication services from the protocol stack;
  - Design a framework that dynamically reconfigures the network protocol stack on every device in the WSN to meet the application run-time requirements;
  - Demonstrate the run-time network communication adaptation feasibility through experiments on a remotely installed testbed;

- Demonstrating a need for designing WSN software as a composition of applications and system services.
  - Implement and evaluate a WSN operating with a data collection application and a system service monitoring the energy-harvesting rate of the WSN devices as well as the rate at which the application's execution consumes the energy;
  - Model the continues system services as a feedback control system;
  - Model the asynchronous execution of the energy-management and the application tasks as a finite state machine;
  - Show how to design a system that autonomously adjusts the application execution, e.g. by adapting the sensing rate to the rate at which energy is consumed and harvested by a WSN device;
  - Demonstrate the feasibility of the proposed WSN system modeling and implementation on an energy neutral sensing system operation;
- Designing and implementing a framework for deploying and testing WSN prototypes.
  - Address the prevalent issues in multi-disciplinary CPS projects which rely on an actual deployment of control systems utilizing sensor and actuator peripherals connected together in a network of low-power wireless embedded devices;
  - Provide software tools that simplify the setup of flexible testbed architectures for relatively low hardware costs;
  - Demonstrate the functionality of the framework on testbed deployments in outdoor and indoor environments, in industry and academia, respectively;
- Designing and implementing a distributed WSN system running with multiple tasks across the network.
  - Introduce programming language for scheduling the execution of tasks across the network;
  - Design and implement a WSN middleware that creates an abstraction of a single system;

– Design and evaluate two network protocols that are used for starting a simultaneous execution of tasks on the WSN and for synchronizing data on all the network devices;

– Demonstrate the performance and functionality of the system using data collection application through experiments on a remote testbed;

7.2 Used Cases

Since the first version of the Swift Fox programming language and the Fennec Fox platform implementation, the software was open-source and publicly available [1]. As a result, the software developed as part of this dissertation was tested and used in academia and industry, and was used to teach low-power wireless sensor networks at Columbia University through semester-long student research projects. The following is the list of projects and places where the software was used.

- **Energy-Harvesting Active Networked Tags (EnHANTs)** is the Columbia University research project collaboration between the Electrical Engineering and the Computer Science departments. This project explores the feasibility and development challenges of designing self-sustainable tags that can be used for tracking objects. These tags are smaller and simpler than typical WSN motes but can do more than traditional RF tags since they can harvest their energy from the environment, process information, and initiate wireless communication on their own.

  The first prototype of the Fennec Fox software was used in the EnHANTs tags. The software included four layered network protocol stack and an integrated communication flow between the application logic and the low-lever Ultra-Wideband Impulse Radio (UWB-IR) driver. The software was used to design, prototype, test, and demonstrate the networking capabilities of the EnHANTS tags. The framework enabled other students to continue the research on the cross-layer system design, networking protocols, and MAC protocols.

  Figure 7.1 shows one of the versions of the EnHANTs tags [2]. On the top, we see a prototype of the UWB-IR radio. The right side shows an energy harvesting module with a solar cell.

[1] SmartCity project website at Columbia University: `http://smartcity.cs.columbia.edu/`

[2] Picture from the EnHANTs demo video: `https://www.youtube.com/watch?v=QMw_Jnv8Cqc` .

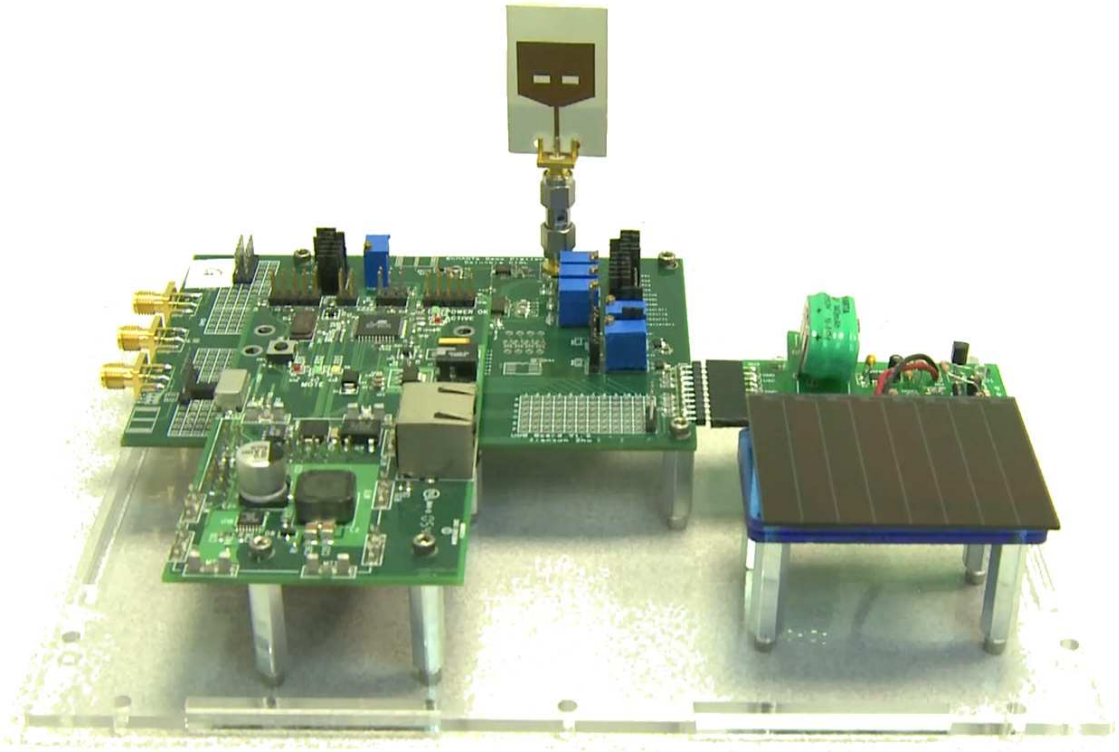


Figure 7.1: A picture of an EnHANTs platform prototype.

The bottom contains a network interface for the firmware reprogramming. The tag prototype runs on the microcontroller from mica2 mote. The microcontroller is programmed with the Fennec Fox firmware and the Swift Fox programs.

The results of this project were presented at academic conferences, totaling in three demos on different conferences [64; 178; 213]. The demo presented at SenSys 2011 received the Best Demo Award. The summary of the EnHANTs project was published in INFOCOM [63].

- **Philips Research North America (PRNA)** is located in Briarcliff Manor, New York. This research center focuses on health-care applications and lighting systems and services. During three summer internships, I was working at Philips on Smart City applications, particularly the intelligent outdoor lighting services.

  At Philips, Fennec Fox software was used to program a WSN deployed outdoor on the light

poles. The light poles were installed across a large parking lot. The WSN motes consisted of Zolertia Z1 motes and various sensors, including motion, sound, and temperature. The motes were running with a firmware periodically sampling sensors and sending their measurements to the network gateway.

The Fennec Fox software enabled the testing of various configurations of the data collection applications. This allowed a better understanding of the events happening on the parking lot, such as presence of people and car movement. Monitoring and detecting of these events led to improvements in the parking lot lighting control, better user experience, and more energy-efficient system operation. The result of these internships was a joint publication at the IEEE International Symposium on Industrial Embedded Systems [184].

- **United Technologies Research Center (UTRC)** is the research division of the United Technologies Corporation located in Hartford, Connecticut. UTRC and Columbia University participate in multidisciplinary NSF Grant Opportunities for Academic Liaison with Industry (GOALI) project, researching methods for designing cyber-physical systems, in particular control systems for high-performance buildings. The goal of the project is to reduce the building energy consumption. In 2013, 40% of the total energy consumption in the United States was used in residential and commercial buildings, which are mostly heated by burning fossil fuels [49].

  The Fennec Fox software was used to provide a network-enabled embedded monitoring of commercial buildings. The software was used in two phases. First, during my visit at UTRC, I demonstrated the software functionality. During this visit, Fennec Fox was used to map and understand the wireless connectivity in a commercial building and to collect preliminary sensor measurements before scaling out the WSN instrumentation. In the second phase, updated software was sent to Hartford, CT, including the tutorials and documentation. Using these resources, UTRC researchers deployed WSN in one of their buildings and were adjusting the WSN functionality by recompiling Swift Fox programs.

- **Student Projects.** Fennec Fox framework and the Swift Fox programming language were used to teach the design of WSN and low-power embedded systems through hands-on experience during multiple research projects with undergraduate and graduate students. Notable

projects include the implementation of the TCP/IP stack for the WSN, an example of industrial sensing systems with an application for remote water level monitoring, the evaluation of the raw sensor measurements in the context of SmartCity applications, and examples of distributed computation within the WSN.

7.3 Avenues of Future Research

In the near future, most low-power devices will likely operate with ARM microcontrollers and microprocessors. Taking advantage of the improved hardware computation resources opens a wide spectrum of new research opportunities in terms of designing new hardware peripherals and new embedded software, including new operating systems. In this dissertation, I often referred to TinyOS and Contiki as the dominant WSN operating systems, and I also indicated the industrial dominance of FreeRTOS. However, just as Linux was designed for a specific architecture, Intel x86, there is a growing need to design an operating system dedicated for the low-power devices running with one of the ARM Cortex-M family microcontrollers. One example of such an attempt is the ARM mbed OS [8], which is a very recent initiative that has the potential to become the future embedded operating system in the WSN and the IoT spaces.

The rise of the Internet-of-Things will continue to dramatically change not only the low-power devices technology, but also the whole field of computer science and related fields. The smart devices, which have already begun penetrating the market and the home environment, are setting up standards and directions for future works. One of the challenges is the heterogeneity of the embedded devices, in terms of both the role they have within the system and the hardware platforms on which they operate. Some aspects of the system heterogeneity will continue to cross the software-hardware boundary, as it has already happened for the case of the multiple different radios that are included on the new WSN platforms and dedicated to different roles implemented at the software level.

In the context of those technological trends, the work and ideas presented in this dissertation can be enhanced with the following research directions:

- Run-time extension and scheduling of new task. The presented system design methodology does not allow changing the firmware running on the WSN devices. Once the tasks, events,

and the network states are defined, at design-time, the WSN continues to operate with those tasks and with the fixed finite state machine model of the system behavior. Although techniques for reprogramming the WSN by disseminating new firmware image are available [70; 87; 130; 208], these techniques have not been tested in the context of the system presented in this dissertation because of their dissemination overhead. One of the challenges in the WSN software upgrades comes from the overhead of disseminating the whole firmware image, which can be on the order of 60KB or more. The other challenge is the difficulty of breaking down the system firmware into independent modules that can be separately upgraded, the issue that was raised by the early industrial adopters of TinyOS [123].

Supporting the run-time addition of new application tasks and a dynamic upgrade of the existing system services requires new software development platforms for the sensor devices themselves. The current implementation of the Fennec Fox with nesC and TinyOS introduces challenges in independently upgrading individual system modules. The source of the problem lies in nesC that optimizes its code before generating a single C-file that is further compiled as a single firmware image. Consequently, TinyOS does not allow to change parts of the system at run-time. Thus, one of the features of the new embedded operating system, which could adapt my system design methodology and enhance it with run-time addition of new tasks, is support for dynamic memory allocation with module loading and unloading. The WSN operating systems that already provide such functionality are SOS [71] and Contiki [41]. Other candidates that are used in industry are FreeRTOS [59] and mbed OS [8].

- Taking advantage of multiple radios. New WSN platforms are designed with multiple radios. UCMote Proton B [193] and Opal [94] have two IEEE 802.15.4-complaint radios operating at 2.4GHz and sub-1GHz frequency bands. The software developed as part of this dissertation work was tested on those platforms. There have been no further investigation or application examples evaluating the same WSN running on platforms with multiple radios and simultaneously executing different tasks using different radios.

  The rise of the IoT devices continues to bring more radio technologies into the WSN world. BLE and WiFi are expected to run at least on some WSN platforms to provide gateway access to other wired networks or cellular, particularly the third generation of the mobile

telecommunication technology (3G). Thus, the radio hardware heterogeneity necessitates orchestrating the computation and communication on the WSN across the different physical channels of the wireless spectrum. These research directions require new network methods to detect different radio technologies and, potentially decide at run-time, which radio to chose to optimize the application performance. There is also a need for computer-aided design (CAD) tools to assist in developing new WSN platforms and, at design-time, deciding which communication technologies to use [122].

- Support 6LoWPAN and the standardized communication platforms. Industry forms alliances for providing the interoperability between low-power devices manufactured by various vendors. The new upcoming standardization efforts are Thread [190] and Z-Wave [173], both adopting the IEEE 802.15.4 [90] and 6LoWPAN [86]. The first stable version of the Fennec Fox framework was supporting the TCP/IP stack. In the fall of 2011, Kshitij Raj Dogra and Sabari Kumar Murugesan implemented Fennec Fox modules providing basic functionality of the IP addressing and TCP communication. This work was concluded with a technical report, *TCP/IP meets Fennec Fox* [3]. However, the IP support was not ported to the newer versions of the Fennec Fox framework. At this moment, the framework does not support the 6LoWPAN standard either.

  Adding 6LoWPAN into a multitasking framework can be used as an exploration into bridging various communication standards. The first benefit of porting 6LoWPAN and standards such as Thread and Z-Wave would be a firmware that can simultaneously operate with different vendors, using different standards. With the WSN design approach presented in this dissertation, different tasks can use various industrial standards, thereby providing both the interoperability as well as bridging services between different standardization domains. The second benefit of adapting 6LoWPAN and industrial standards would be addressing the run-time performance gaps in the standardization efforts. RPL [200], which is used by some of the standards, already requires modifications [44; 158]. Adding new amendments into the standard takes effort and time and it does not guarantee achieving the desired operation performance. Thus, to meet the new products application requirements, some vendors will

[3] Project Report: `http://www.cs.columbia.edu/~msz/projects/2011-Fall-TCPIP/final_report.pdf`

be forced to adapt the state of the research protocols, which are in-house designed or publicly published but not standardized. In such situations, some tasks can be scheduled to run with the standardized protocols, while others, that have special requirements, can run on customized protocols, thus enabling industry to release new products and features that cannot afford to wait for the standardization process.

Acronyms

6LoWPAN IPv6 over low power wireless personal area networks

ADC analog-to-digital converter

ASCII American standard code for information interchange

BCP broadcast control process

BEDS best effort data synchronization

BLE bluetooth low energy

CCA clear channel assessment

CM control message

CoAP constrained application protocol

CPS cyber-physical system

CRC cyclic redundancy check

CSMA carrier sense multiple access

CTP collection tree protocol

DARPA defense advanced research projects agency

EED estimate the end of the dissemination

EHA energy harvesting actuation

EHS energy harvesting sensing

ENSSys energy neutral sensing systems

FSM finite-state machine

GPIO general-purpose input/output

HTML hypertext markup language

HTTP hypertext transfer protocol

HVAC heating, ventilating, and air conditioning

I^{2}C inter-integrated circuit

IEEE institute of electrical and electronics engineers

IETF Internet engineering task force

IoT Internet of things

LED light-emitting diode

LPL low-power listening

LPWN low-power wireless network

MAC media access control

MMU memory management unit

MPU memory protection unit

NSF national science foundation

OLSR optimized link state routing

OTF open testbed framework

PC personal computer

PDR packet delivery rate

PIR passive infrared sensor

PNP parasite network protocol

RAM random-access memory

REST representational state transfer

ROM read-only memory

RPL IPv6 routing protocol for low-power and lossy networks

RTOS real-time operating system

SPI serial peripheral interface

SQL structured query language

TDMA time division multiple access

UART universal asynchronous receiver/transmitter

USB universal serial bus

URL uniform resource identifier

UWB-IR ultra wide-band impulse radio

WPAN wireless personal area network

WSN wireless sensor network

Bibliography

[1] Norman Abramson. THE Aloha SYSTEM: another alternative for computer communications. In *Proc. of the AFIPS Conf.*, pages 281–285, November 1970.

[2] Cedric Adjih, Ichrak Amdouni, Hana Baccouch, and Antonia Masucci. Scientific Experiments with the Large Scale Open Testbed IoT-LAB: Broadcast with Network Coding. In *Proc. of the IEEE MASS*, October 2014.

[3] Robert Adler, Mick Flanigan, Jonathan Huang, Ralph Kling, Nandakishore Kushalnagar, Lama Nachman, Chieh-Yih Wan, and Mark Yarvis. Intel Mote 2: An Advanced Platform for Demanding Sensor Network Applications. In *Proc. of the ACM SenSys Conf.*, pages 298–298, November 2005.

[4] Y. Afsar, J. Sarik, M. Gorlatova, G. Zussman, and I. Kymissis. Evaluating photovoltaic performance indoors. In *Proc. of the Photovoltaic Specialists Conf.*, pages 1948–1951, June 2012.

[5] Daniel Van Den Akker and Chris Blondia. MultiMAC: A Multiple MAC Network Stack Architecture for TinyOS. In *Proc. in the ICCCN Conf.*, pages 1–5, August 2012.

[6] Massimo Alioto. Ultra-Low Power VLSI Circuit Design Demystified and Explained: A Tutorial. *Circuits and Systems I: Regular Papers, IEEE Transactions on*, 59(1):3–29, jan 2012.

[7] ARM. Cortex-M Series: Scalable and Low-Power Technology for any Embedded Market. [Online] `http://www.arm.com/products/processors/cortex-m/`, 2014.

[8] ARM mbed. ARM mbed IoT Device Platform. [Online] `https://mbed.org/technology/os/`, 2014.

[9] A. Arora, E. Ertin, R. Ramnath, M. Nesterenko, and W. Leal. Kansei: A High-Fidelity Sensing Testbed. In *Internet Computing, IEEE*, volume 10, pages 35–47, March 2006.

[10] Kevin Ashton. That 'Internet of Things' Thing. [Online] `http://www.rfidjournal.com/articles/view?4986`, June 2009.

[11] Atmel. 8-bit Atmel Microcontroller with 128KBytes In-System Programmable Flash. [Online] `http://www.atmel.com/images/doc2467.pdf`.

[12] Ravneet Bajwa, Ram Rajagopal, Erdem Coleri, Pravin Varaiya, and Christopher Flores. In-pavement Wireless Weigh-in-motion. In *Proc. of the IPSN Conf.*, pages 103–114, April 2013.

[13] Rahul Balani, Akhilesh Singhania, Chih-Chieh Han, and Mani Srivastava. ViRe: Virtual Reconfiguration Framework for Embedded Processing in Distributed Image Sensors. In *APRES Work.*, April 2008.

[14] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, and Richard Han. MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms. *Mob. Netw. Appl.*, 10(4):563–579, August 2005.

[15] Bill Curtis, ARM. Standards for Constrained IoT Devices. [Online] `https://www.youtube.com/watch?v=3WDQepWIs9A`, 2014.

[16] Bluetooth. Bluetooth Smart Technology: Powering the Internet of Things. [Online] `http://www.bluetooth.com/Pages/Bluetooth-Smart.aspx`, 2015.

[17] Carlo Alberto Boano, Marco Zúñiga, James Brown, Utz Roedig, Chamath Keppitiyagama, and Kay Römer. TempLab: A Testbed Infrastructure to Study the Impact of Temperature on Wireless Sensor Networks. In *Proc. of the IPSN Conf.*, pages 95–106, April 2014.

[18] Michael Buettner, Gary V. Yee, Eric Anderson, and Richard Han. X-MAC: a short preamble MAC protocol for duty-cycled wireless sensor networks. In *Proc. of the ACM SenSys Conf.*, pages 307–320, November 2006.

[19] Maxim Buevich, Dan Schnitzer, Tristan Escalada, Arthur Jacquiau-Chamski, and Anthony Rowe. A System for Fine-Grained Remote Monitoring, Control and Pre-Paid Electrical Service in Rural Microgrids. In *Proc. of the IPSN Conf.*, pages 1–12, April 2014.

[20] Nicolas Burri, Pascal von Rickenbach, and Rogert Wattenhofer. Dozer: Ultra-low Power Data Gathering in Sensor Networks. In *Proc. of the IPSN Conf.*, pages 450–459, April 2007.

[21] Qing Cao, Tarek Abdelzaher, John Stankovic, and Tian He. The LiteOS Operating System: Towards Unix-Like Abstractions for Wireless Sensor Networks. In *Proc. of the IPSN Conf.*, pages 233–244, April 2008.

[22] M. Ceriotti, M. Corra, L. D'Orazio, R. Doriguzzi, D. Facchin, S.T. Guna, G.P. Jesi, R. Lo Cigno, L. Mottola, A.L. Murphy, M. Pescalli, G.P. Picco, D. Pregnolato, and C. Torghele. Is there light at the ends of the tunnel? Wireless sensor networks for adaptive lighting in road tunnels. In *Proc. of the IPSN Conf.*, pages 187–198, April 2011.

[23] Alberto Cerpa, Jennifer L. Wong, Miodrag Potkonjak, and Deborah Estrin. Temporal properties of low power wireless links: modeling and implications on multi-hop routing. In *Proc. of the MobiHoc Symp.*, pages 414–425, May 2005.

[24] Hojung Cha, Sukwon Choi, Inuk Jung, Hyoseung Kim, Hyojeong Shin, Jaehyun Yoo, and Chanmin Yoon. RETOS: Resilient, Expandable, and Threaded Operating System for Wireless Sensor Networks. In *Proc. of the IPSN Conf.*, pages 148–157, April 007.

[25] Yu-Ting Chen, Ting-Chou Chien, and Pai H. Chou. Enix: a lightweight dynamic operating system for tightly constrained wireless sensor platforms. In *Proc. of the ACM SenSys Conf.*, pages 183–196, November 2010.

[26] Yun Cheng, Xiucheng Li, Zhijun Li, Shouxu Jiang, Yilong Li, Ji Jia, and Xiaofan Jiang. AirCloud: A Cloud-based Air-quality Monitoring System for Everyone. In *Proc. of the ACM SenSys Conf.*, pages 251–265, November 2014.

[27] U.M. Colesanti, S. Santini, and A. Vitaletti. DISSense: An adaptive ultralow-power communication protocol for wireless sensor networks. In *Proc. of IEEE DCOSS Conf.*, pages 1–10, June 2011.

[28] Paolo Costa, Luca Mottola, Amy L. Murphy, and Gian Pietro Picco. Programming Wireless Sensor Networks with the TeenyLime Middleware. In *Proc. of the Middleware Conf.*, pages 429–449, December 2007.

[29] Geoff Coulson, Barry Porter, Ioannis Chatzigiannakis, Christos Koninis, Stefan Fischer, Dennis Pfisterer, Daniel Bimschas, Torsten Braun, Philipp Hurni, Markus Anwander, Gerald Wagenknecht, Sándor P. Fekete, Alexander Kröller, and Tobias Baumgartner. Flexible experimentation in wireless sensor networks. *Commun. ACM*, 55(1):82–90, January 2012.

[30] David Culler. TinyOS - Time to ROLL. [Online] `http://vimeo.com/3264574`, February 2009.

[31] David Culler, Prabal Dutta, Cheng Tien Ee, Rodrigo Fonseca, Jonathan Hui, Philip Levis, Joseph Polastre, Scott Shenker, Ion Stoica, Gilman Tolle, and Jerry Zhao. Towards a Sensor Network Architecture: Lowering the Waistline. In *Proc. of the HotOS-X Conf.*, pages 139–144, June 2005.

[32] David Culler, Deborah Estrin, and Mani Srivastava. Guest Editors' Introduction: Overview of Sensor Networks. *Computer*, 37:41–49, August 2004.

[33] D. Datla, Xuetao Chen, T. Tsou, S. Raghunandan, S.M. Shajedul Hasan, J.H. Reed, C.B. Dietrich, T. Bose, B. Fette, and J. Kim. Wireless distributed computing: a survey of research challenges. *Communications Magazine, IEEE*, 50(1):144–152, jan 2012.

[34] Stephen Dawson-Haggerty, Xiaofan Jiang, Gilman Tolle, Jorge Ortiz, and David Culler. sMAP: A Simple Measurement and Actuation Profile for Physical Information. In *Proc. of the ACM SenSys Conf.*, pages 197–210, November 2010.

[35] Manjunath Doddavenkatappa, Mun Choon Chan, and A.L Ananda. Indriya: A Low-Cost, 3D Wireless Sensor Network Testbed. In *TRIDENTCOM*, pages 302–316, April 2011.

[36] Manjunath Doddavenkatappa, Mun Choon Chan, and Ben Leong. Splash: Fast Data Dissemination with Constructive Interference in Wireless Sensor Networks. In *Proc. of the NSDI Conf.*, pages 269–282, April 2011.

[37] Ann Dooley. Intel Brings Out 8-Bit MPU Featuring 16-Bit Architecture. 13(20):72–72, May 1979.

[38] Nildo dos Santos Ribeiro Júnior, MarcosA.M. Vieira, LuizF.M. Vieira, and Omprakash Gnawali. CodeDrip: Data Dissemination Protocol with Network Coding for Wireless Sensor Networks. In Bhaskar Krishnamachari, AmyL. Murphy, and Niki Trigoni, editors, *Wireless Sensor Networks*, volume 8354 of *Lecture Notes in Computer Science*, pages 34–49, 2014.

[39] Adam Dunkels. Full TCP/IP for 8 Bit Architectures. In *Proc. of the MobiSys Conf.*, pages 85–98, May 2003.

[40] Adam Dunkels. The ContikiMAC Radio Duty Cycling Protocol. Technical Report. Swedish Institute of Computer Science, 2011.

[41] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a Lightweight and Flexible Operating System for Tiny Networked Sensors. In *Proc. of the IEEE LCN Conf.*, pages 455–462, November 2004.

[42] Adam Dunkels, Fredrik Österlind, and Zhitao He. An adaptive communication architecture for wireless sensor networks. In *Proc. of the ACM SenSys Conf.*, pages 335–349, November 2007.

[43] Adam Dunkels, Oliver Schmidt, Thiemo Voigt, and Muneeb Ali. Protothreads: Simplifying Event-driven Programming of Memory-constrained Embedded Systems. In *Proc. of the ACM SenSys Conf.*, pages 29–42, November 2006.

[44] Simon Duquennoy, Olaf Landsiedel, and Thiemo Voigt. Let the Tree Bloom: Scalable Opportunistic Routing with ORPL. In *Proc. of the ACM SenSys Conf.*, pages 2:1–2:14, November 2013.

[45] Prabal Dutta, Stephen Dawson-Haggerty, Yin Chen, Chieh-Jan Mike Liang, and Andreas Terzis. A-MAC: A Versatile and Efficient Receiver-initiated Link Layer for Low-power Wireless. *ACM Trans. Sen. Netw.*, 8(4):30:1–30:29, September 2012.

[46] Prabal Dutta, Jonathan Hui, Jaein Jeong, Sukun Kim, Cory Sharp, Jay Taneja, Gilman Tolle, Kamin Whitehouse, and David Culler. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proc. of the IPSN Conf.*, pages 407–415, April 2006.

[47] Prabal Dutta, Răzvan Musăloiu-e, Ion Stoica, and Andreas Terzis. Wireless ACK collisions not considered harmful. In *ACM HotNets Work.*, October 2008.

[48] Prabal Dutta, Jay Taneja, Jaein Jeong, Xiaofan Jiang, and David Culler. A building block approach to sensornet systems. In *Proc. of the ACM SenSys Conf.*, pages 267–280, November 2008.

[49] Energy Information Administration. How much energy is consumed in residential and commercial buildings in the United States? [Online] `http://www.eia.gov/tools/faqs/faq.cfm?id=86&t=1`, June 2014.

[50] EnOcean. General Purpose Sensor Transmitter Module STM 31x/STM 31xC, April 2013.

[51] Varick L. Erickson, Stefan Achleitner, and Alberto E. Cerpa. POEM: Power-efficient Occupancy-based Energy Management System. In *Proc. of the IPSN Conf.*, pages 203–216, April 2013.

[52] Deborah Estrin, Ramesh Govindan, John Heidemann, and Satish Kumar. Next Century Challenges: Scalable Coordination in Sensor Networks. In *Proc. of the MobiCom Conf.*, pages 263–270, August 1999.

[53] Anand Eswaran, Anthony Rowe, and Raj Rajkumar. Nano-RK: An Energy-Aware Resource-Centric RTOS for Sensor Networks. In *Proc. of the RTSS*, pages 256–265, December 2005.

[54] Federico Ferrari, Marco Zimmerling, Luca Mottola, and Lothar Thiele. Low-power Wireless Bus. In *Proc. of the ACM SenSys Conf.*, pages 1–14, November 2012.

[55] Federico Ferrari, Marco Zimmerling, Lothar Thiele, and Olga Saukh. Efficient network flooding and time synchronization with Glossy. In *Proc. of the IPSN Conf.*, pages 73–84, April 2011.

[56] Roy T. Fielding and Richard N. Taylor. Principled Design of the Modern Web Architecture. In *Proc. of the ICSE Conf.*, pages 407–416, June 2000.

[57] Lisa Fleisher. British Chip Designer ARM's Profit Rises. *The Wall Street Journal*, October 14, 2014.

[58] Romain Fontugne, Jorge Ortiz, Nicolas Tremblay, Pierre Borgnat, Patrick Flandrin, Kensuke Fukuda, David Culler, and Hiroshi Esaki. Strip, Bind, and Search: A Method for Identifying Abnormal Energy Consumption in Buildings. In *Proc. of the IPSN Conf.*, pages 129–140, April 2013.

[59] FreeRTOS. [Online] `http://www.freertos.org/`, 2014.

[60] Saurabh Ganeriwal, Ram Kumar, and Mani B. Srivastava. Timing-sync Protocol for Sensor Networks. In *Proc. of the ACM SenSys Conf.*, pages 138–149, November 2003.

[61] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesC Language: A Holistic Approach to Networked Embedded Systems. In *Proc. of the PLDI*, pages 1–11, May 2003.

[62] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection tree protocol. In *Proc. of the ACM SenSys Conf.*, pages 1–14, November 2009.

[63] M. Gorlatova, R. Margolies, J. Sarik, G. Stanje, Jianxun Zhu, B. Vigraham, M. Szczodrak, L. Carloni, P. Kinget, I. Kymissis, and G. Zussman. Prototyping Energy Harvesting Active Networked Tags (EnHANTs). In *Proc. IEEE INFOCOM'13 mini-conference*, pages 585–589, April 2013.

[64] Maria Gorlatova, Zainab Noorbhaiwala, Abraham Skolnik, John Sarik, Michael Zapas, Marcin Szczodrak, Jiasi Chen, Luca Carloni, Peter Kinget, Ioannis Kymissis, Dan Rubenstein, and Gil Zussman. Prototyping Energy Harvesting Active Networked Tags: Phase II MICA Mote-based Devices. In *MobiCom'10*, September 2010.

[65] Maria Gorlatova, John Sarik, Guy Grebla, Mina Cong, Ioannis Kymissis, and Gil Zussman. Movers and Shakers: Kinetic Energy Harvesting for the Internet of Things. *Proc. of the SIGMETRICS*, pages 407–419, June 2013.

[66] Maria Gorlatova, Aya Wallwater, and Gil Zussman. Networking Low-Power Energy Harvesting Devices: Measurements and Algorithms. In *Proc. IEEE INFOCOM'11*, pages 1602–1610, April 2011.

[67] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Macroprogramming Wireless Sensor Networks Using Kairos. In *Proc. of the IEEE DCOSS Conf.*, pages 126–140, June 2005.

[68] V. Gupta, Junsung Kim, A. Pandya, K. Lakshmanan, R. Rajkumar, and E. Tovar. Nano-CF: A coordination framework for macro-programming in Wireless Sensor Networks. In *Proc. of the SECONN Conf.*, pages 467–475, June 2011.

[69] Vikram Gupta, Eduardo Tovar, Luis Miguel Pinho, Junsung Kim, Karthik Lakshmanan, and Ragunathan(Raj) Rajkumar. sMapReduce: A Programming Pattern for Wireless Sensor Networks. In *Proc. of the SESENA Work.*, pages 37–42, May 2011.

[70] Andrew Hagedorn, David Starobinski, and Ari Trachtenberg. Rateless Deluge: Over-the-Air Programming of Wireless Sensor Networks Using Random Linear Codes. In *Proc. of the IPSN Conf.*, pages 457–466, April 2008.

[71] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A Dynamic Operating System for Sensor Nodes. In *Proc. of the MobiSys Conf.*, pages 163–176, June 2005.

[72] Dong Han and Omprakash Gnawali. Understanding Desktop Energy Footprint in an Academic Computer Lab. In *Proc. of the GreenCom Conf.*, pages 541–548, November 2012.

[73] Abhiman Hande, Todd Polk, William Walker, and Dinesh Bhatia. Indoor solar energy harvesting for sensor network router nodes. *Microprocessors and Microsystems*, 31(6):420–432, 2007.

[74] Marcus Handte, Stephan Wagner, Gregor Schiele, Christian Becker, and Pedro José Marrón. The BASE Plug-in Architecture - Composable Communication Support for Pervasive Systems. In *Proc. of the ICPS Conf.*, page 443, July 2010.

[75] Vlado Handziski, Andreas Köpke, Andreas Willig, and Adam Wolisz. TWIST: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proc. of Work. REALMAN*, pages 63–70, May 2006.

[76] Vlado Handziski, Joseph Polastre, Jan-Hinrich Hauer, Cory Sharp, Adam Wolisz, and David Culler. Flexible hardware abstraction for wireless sensor networks. In *Proc. of EWSN Conf.*, pages 145–157, February 2005.

[77] Tyler Harms, Sahra Sedigh, and Filippo Bastianini. Structural Health Monitoring of Bridges Using Wireless Sensor Networks. *Instrumentation Measurement Magazine, IEEE*, 13(6):14–18, December 2010.

[78] H. Hellbruck, M. Pagel, A. Kroller, D. Bimschas, D. Pfisterer, and S. Fischer. Using and operating wireless sensor network testbeds with WISEBED. In *Proc. of the Med-Hoc-Net Work.*, pages 171–178, June 2011.

[79] Josiah Hester, Timothy Scott, and Jacob Sorber. Ekho: Realistic and Repeatable Experimentation for Tiny Energy-harvesting Sensors. In *Proc. of the ACM SenSys Conf.*, pages 1–15, November 2014.

[80] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System Architecture Directions for Networked Sensors. In *Proc. of the ACM SenSys Conf.*, pages 93–104, November 2000.

[81] Jason L. Hill and David E. Culler. Mica: A Wireless Platform for Deeply Embedded Networks. *IEEE Micro*, 22(6):12–24, November 2002.

[82] Timothy W. Hnat, Erin Griffiths, Ray Dawson, and Kamin Whitehouse. Doorjamb: unobtrusive room-level tracking of people in homes using doorway sensors. In *Proc. of the ACM SenSys Conf.*, pages 309–322, November 2012.

[83] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. MacroLab: A Vector-based Macroprogramming Framework for Cyber-physical Systems. In *Proc. of the ACM SenSys Conf.*, pages 225–238, November 2008.

[84] Wen Hu, Nirupama Bulusu, Chun Tung Chou, Sanjay Jha, Andrew Taylor, and Van Nghia Tran. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *Proc. of the IPSN Conf.*, pages 503–508, April 2005.

[85] William Huang, Ye-Sheng Kuo, Pat Pannuto, and Prabal Dutta. Opo: A Wearable Sensor for Capturing High-fidelity Face-to-face Interactions. In *Proc. of the ACM SenSys Conf.*, pages 61–75, November 2014.

[86] J. Hui and P. Thubert. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. Internet Engineering Task Force (IETF), Request for Comments (RFC): 6282, September 2011.

[87] Jonathan W Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the ACM SenSys Conf.*, pages 81–94, November 2004.

[88] J.W. Hui and D.E. Culler. IPv6 in Low-Power Wireless Networks. *Proc. of the IEEE*, 98(11):1865–1878, November 2010.

[89] IBM. IBM (Information Systems Division, Entry Systems Business) Press Release, August 12, 1981: Personal Computer Announced by IBM. [Online] `http://www-03.ibm.com/ibm/history/documents/pdf/pcpress.pdf`, August 1981.

[90] Institute of Electrical and Electronics Engineers. IEEE 802.15: Wireless Personal Area Networks (PANs). [Online] `http://standards.ieee.org/about/get/802/802.15.html`.

[91] Mayank Jain, Jung Il Choi, Taemin Kim, Dinesh Bharadia, Siddharth Seth, Kannan Srinivasan, Philip Levis, Sachin Katti, and Prasun Sinha. Practical, Real-time, Full Duplex Wireless. In *Proc. of MobiCom Conf.*, pages 301–312, September 2011.

[92] Fatemeh Jalali, Rob Ayre, Arun Vishwanath, Kerry Hinton, Tansu Alpcan, and Rod Tucker. Energy Consumption of Content Distribution from Nano Data Centers Versus Centralized Data Centers. *SIGMETRICS Perform. Eval. Rev.*, 42(3):49–54, December 2014.

[93] Vena Jelicic, Michele Magno, Davide Brunelli, Giacomo Paci, and Luca Benini. Context-Adaptive Multimodal Wireless Sensor Network for Energy-Efficient Gas Monitoring. *Sensors Journal, IEEE*, 13(1):328–338, January 2013.

[94] Raja Jurdak, Kevin Klues, Brano Kusy, Christian Richter, Koen Langendoen, and Michael Briünig. Opal: A Multiradio Platform for High Throughput Wireless Sensor Networks. *Embedded Systems Letters*, 3(4):121–124, 2011.

[95] Ankur Kamthe, Miguel Á Carreira-Perpiñán, and Alberto E. Cerpa. Improving Wireless Link Simulation Using Multilevel Markov Models. *ACM Trans. Sen. Netw.*, 10(1):17:1–17:28, December 2013.

[96] Holger Karl and Andreas Willig. *Protocols and Architectures for Wireless Sensor Networks*. John Wiley & Sons, 2005.

[97] Maria Kazandjieva, Chinmayee Shah, Ewen Cheslack-Postava, Behram Mistree, and Philip Levis. System Architecture Support for Green Enterprise Computing. In *Proc. of the IGCC Conf.*, November 2014.

[98] Bryce Kellogg, Aaron Parks, Shyamnath Gollakota, Joshua R. Smith, and David Wetherall. Wi-fi Backscatter: Internet Connectivity for RF-powered Devices. In *Proc. of the SIGCOMM Conf.*, pages 607–618, August 2014.

[99] Gyouho Kim, Yoonmyung Lee, Zhiyoong Foo, P. Pannuto, Ye sheng Kuo, B. Kempke, M.H. Ghaed, Suyoung Bang, Inhee Lee, Yejoong Kim, Seokhyeon Jeong, P. Dutta, D. Sylvester, and D. Blaauw. A millimeter-scale wireless imaging system with continuous motion detection and energy harvesting. In *Proc. of the VLSI Symp.*, pages 1–2, June 2014.

[100] Sukun Kim, Rodrigo Fonseca, Prabal Dutta, Arsalan Tavakoli, David Culler, Philip Levis, Scott Shenker, and Ion Stoica. Flush: a reliable bulk transport protocol for multihop wireless networks. In *Proc. of the ACM SenSys Conf.*, pages 351–365, November 2007.

[101] Younghun Kim, Thomas Schmid, Zainul M. Charbiwala, Jonathan Friedman, and Mani B. Srivastava. NAWMS: Nonintrusive Autonomous Water Monitoring System. In *Proc. of the ACM SenSys Conf.*, pages 309–322, November 2008.

[102] Ralph Kling, Robert Adler, Jonathan Huang, Vincent Hummel, and Lama Nachman. Intel Mote: Using Bluetooth in Sensor Networks. In *Proc. of the ACM SenSys Conf.*, pages 318–318, November 2004.

[103] Kevin Klues, Chieh-Jan Mike Liang, Jeongyeup Paek, Răzvan Musăloiu-E, Philip Levis, Andreas Terzis, and Ramesh Govindan. TOSThreads: Thread-safe and Non-invasive Preemption in TinyOS. In *Proc. of the ACM SenSys Conf.*, pages 127–140, November 2009.

[104] JeongGil Ko, Joakim Eriksson, Nicolas Tsiftes, Stephen Dawson-Haggerty, Jean-Philippe Vasseur, Mathilde Durvy, Andreas Terzis, Adam Dunkels, and David Culler. Beyond interoperability: pushing the performance of sensor network IP stacks. In *Proc. of the ACM SenSys Conf.*, pages 1–11, November 2011.

[105] JeongGil Ko, Kevin Klues, Christian Richer, Wanja Hofer, Branislav Kusy, Michael Bruenig, Thomas Schmid, Qiang Wang, Prabal Dutta, and Andreas Terzis. Low Power or High Performance? A Tradeoff Whose Time Has Come (and Nearly Gone). In *Proc. of the EWSN Conf.*, pages 98–114, February 2012.

[106] JeongGil Ko, Chenyang Lu, Mani B. Srivastava, John A. Stankovic, Andreas Terzis, and Matt Welsh. Wireless Sensor Networks for Healthcare. *Proc. of the IEEE*, 98(11):1947–1960, November 2010.

[107] Matthias Kovatsch, Simon Duquennoy, and Adam Dunkels. A Low-Power CoAP for Contiki. In *Proc. of the MASS Conf.*, pages 855–860, October 2011.

[108] Pratyush Kumar, Dip Goswami, Samarjit Chakraborty, Anuradha Annaswamy, Kai Lampka, and Lothar Thiele. A hybrid approach to cyber-physical systems verification. In *Proc. of the Design Automation Conf.*, pages 688–696, June 2012.

[109] Rajnish Kumar, Matthew Wolenetz, Bikash Agarwalla, JunSuk Shin, Phillip Hutto, Arnab Paul, and Umakishore Ramachandran. DFuse: A Framework for Distributed Data Fusion. In *Proc. of the ACM SenSys Conf.*, pages 114–125, November 2003.

[110] Ye-Sheng Kuo, Pat Pannuto, Gyouho Kim, Zhi Yoong Foo, Inhee Lee, Ben Kempke, Prabal Dutta, David Blaauw, and Yoonmyung Lee. MBus: A 17.5 pJ/bit Portable Interconnect Bus

for Millimeter-Scale Sensor Systems with 8 nW Standby Power. In *Proc. on the CICC Conf.*, pages 1–4, September 2014.

[111] Branislav Kusy, David Abbott, Christian Richter, Cong Huynh, Mikhail Afanasyev, Wen Hu, Michael Brünig, Diethelm Ostry, and Raja Jurdak. Radio Diversity for Reliable Communication in Sensor Networks. *ACM Trans. Sen. Netw.*, 10(2):32:1–32:29, January 2014.

[112] Olaf Landsiedel, Federico Ferrari, and Marco Zimmerling. Chaos: Versatile and Efficient All-to-all Data Sharing and In-network Processing at Scale. In *Proc. of the ACM SenSys Conf.*, pages 1:1–1:14, November 2013.

[113] Olaf Landsiedel, Euhanna Ghadimi, Simon Duquennoy, and Mikael Johansson. Low Power, Low Delay: Opportunistic Routing Meets Duty Cycling. In *Proc. of the IPSN Conf.*, pages 185–196, April 2012.

[114] Sei Ping Lau, Geoff V. Merrett, and Neil M. White. Energy-efficient street lighting through embedded adaptive intelligence. In *Proc. of ICALT Conf.*, pages 53–58, May 2013.

[115] Edward A. Lee. Cyber-Physical Systems - Are Computing Foundations Adequate? October 2006.

[116] Edward A. Lee. Cyber Physical Systems: Design Challenges. In *Proc. of the IEEE ISORC Symp.*, pages 363–369, May 2008.

[117] HyungJune Lee, Alberto Cerpa, and Philip Levis. Improving Wireless Simulation Through Noise Modeling. In *Proc. of the IPSN Conf.*, pages 21–30, April 2007.

[118] Sang Hyuk Lee, Soobin Lee, Heecheol Song, and Hwang-Soo Lee. Wireless sensor network design for tactical military applications : Remote large-scale environments. In *Proc. of the MILCOM Conf.*, pages 1–7, October 2009.

[119] Yoonmyung Lee, Suyoung Bang, Inhee Lee, Yejoong Kim, Gyouho Kim, M.H. Ghaed, P. Pannuto, P. Dutta, D. Sylvester, and D. Blaauw. A Modular 1 mm^3 Die-Stacked Sensing Platform With Low Power I^2C Inter-Die Communication and Multi-Modal Energy Harvesting. *Solid-State Circuits, IEEE Journal of*, 48(1):229–243, jan 2013.

[120] Stanisław Lem. *Cyberiada*. Wydawnictwo Literackie, 1965. In Polish. Translated to English by Michale Kandel in 1974.

[121] Christoph Lenzen, Philipp Sommer, and Roger Wattenhofer. Optimal Clock Synchronization in Networks. In *Proc. of the ACM SenSys Conf.*, pages 225–238, November 2009.

[122] Francesco Leonardi, Alessandro Pinto, and Luca P. Carloni. Synthesis of Distributed Execution Platforms for Cyber-Physical Systems with Applications to High-Performance Buildings. In *Proc. of the ICCPS Conf.*, pages 215–224, 2011.

[123] Philip Levis. Experiences from a Decade of TinyOS Development. In *Proc. of the OSDI Symp.*, pages 207–220, October 2012.

[124] Philip Levis and David Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Proc. of the ASPLOS Conf.*, pages 85–95, October 2002.

[125] Philip Levis and David Gay. *TinyOS Programming*. Cambridge University Press, New York, NY, USA, 1st edition, 2009.

[126] Philip Levis, Nelson Lee, Matt Welsh, and David Culler. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of the ACM SenSys Conf.*, pages 126–137, November 2003.

[127] Philip Levis, Sam Madden, Joseph Polastre, Robert Szewczyk, Kamin Whitehouse, Alec Woo, David Gay, Jason Hill, Matt Welsh, Eric Brewer, and David Culler. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–144, November 2004.

[128] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: A Self-regulating Algorithm for Code Propagation and Maintenance in Wireless Sensor Networks. In *Proc. of the NSDI Conf.*, pages 15–28, April 2004.

[129] Qiang Li, Dong Han, Omprakash Gnawali, Philipp Sommer, and Branislav Kusy. Twonet - Large-Scale Wireless Sensor Network Testbed with Dual-Radio Nodes. In *Proc. of the ACM SenSys Conf.*, pages 1–2, November 2013.

[130] Weijia Li, Youtao Zhang, and Bruce Childers. MCP: An Energy-Efficient Code Distribution Protocol for Multi-Application WSNs. In *Proc. of the IEEE DCOSS Conf.*, pages 259–272, June 2009.

[131] Yu-Te Liao, Huanfen Yao, A. Lingley, B. Parviz, and B.P. Otis. A 3-μW CMOS Glucose Sensor for Wireless Contact-Lens Tear Glucose Monitoring. *Solid-State Circuits, IEEE Journal of*, 47(1):335–344, jan 2012.

[132] Roman Lim, Federico Ferrari, Marco Zimmerling, Christoph Walser, Philipp Sommer, and Jan Beutel. FlockLab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proc. of the IPSN Conf.*, pages 153–166, April 2013.

[133] Kaisen Lin and Philip Levis. Data Discovery and Dissemination with DIP. In *Proc. of the IPSN Conf.*, pages 433–444, April 2008.

[134] Xiaojun Lin and Ness B. Shroff. The impact of imperfect scheduling on cross-layer congestion control in wireless networks. *IEEE/ACM Trans. Netw.*, 14(2):302–315, April 2006.

[135] Guojin Liu, Rui Tan, Ruogu Zhou, Guoliang Xing, Wen-Zhan Song, and Jonathan M. Lees. Volcanic Earthquake Timing Using Wireless Sensor Networks. In *Proc. of the IPSN Conf.*, pages 91–102, April 2013.

[136] Vincent Liu, Aaron Parks, Vamsi Talla, Shyamnath Gollakota, David Wetherall, and Joshua R. Smith. Ambient backscatter: wireless communication out of thin air. In *Proc. of the ACM SIGCOMM*, pages 39–50, August 2013.

[137] Konrad Lorincz, Bor-rong Chen, Jason Waterman, Geoff Werner-Allen, and Matt Welsh. Resource Aware Programming in the Pixie OS. In *Proc. of the ACM SenSys Conf.*, pages 211–224, November 2008.

[138] Jiakang Lu, Tamim Sookoor, Vijay Srinivasan, Ge Gao, Brian Holben, John Stankovic, Eric Field, and Kamin Whitehouse. The smart thermostat: using occupancy sensors to save energy in homes. In *Proc. of the ACM SenSys Conf.*, pages 211–224, November 2010.

[139] Jiakang Lu and Kamin Whitehouse. Flash Flooding: Exploiting the Capture Effect for Rapid Flooding in Wireless Sensor Networks. In *Proc. of the INFOCOM Conf.*, pages 2491–2499, April 2009.

[140] Samuel R. Madden, Michael J. Franklin, Joseph M. Hellerstein, and Wei Hong. TinyDB: An Acquisitional Query Processing System for Sensor Networks. *ACM Trans. Database Syst.*, 30(1):122–173, March 2005.

[141] Rahul Mangharam, Anthony Rowe, and Raj Rajkumar. FireFly: A Cross-layer Platform for Real-time Embedded Wireless Networks. *Real-Time Syst.*, 37(3):183–231, December 2007.

[142] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *Proc. of the ACM SenSys Conf.*, pages 39–49, November 2004.

[143] Pedro José Marrón, , Andreas Lachenmann, Daniel Minder, Jörg Hähner, Robert Sauter, and Kurt Rothermel. TinyCubus: a flexible and adaptive framework sensor networks. In *Proc. of the EWSN Conf.*, pages 278–289, January 2005.

[144] Tommaso Melodia, Mehmet C. Vuran, and Dario Pompili. The State of the Art in Cross-layer Design for Wireless Sensor Networks. In *Proc. of the EURO-NGI Conf.*, pages 78–92, April 2006.

[145] Memsic. Lotus: High-Performance Wireless Sensor Network Platform. [Online] `http://www.memsic.com/userfiles/files/Datasheets/WSN/6020-0705-01_A_LOTUS.pdf`, 2014.

[146] Memsic. MicaZ: Wireless Measurement System. [Online] `http://www.memsic.com/userfiles/files/Datasheets/WSN/6020-0060-04-B_MICAz.pdf`, 2014.

[147] Memsic. TelosB: TelosB Mote Platform. [Online] `http://www.memsic.com/userfiles/files/Datasheets/WSN/6020-0094-02_B_TELOSB.pdf`, 2014.

[148] Lucas D. P. Mendes and Joel J.P.C. Rodrigues. Review: A Survey on Cross-layer Solutions for Wireless Sensor Networks. *J. Netw. Comput. Appl.*, 34(2):523–534, March 2011.

[149] P.P. Mercier, D.C. Daly, M. Bhardwaj, D.D. Wentzloff, F.S. Lee, and A.P. Chandrakasan. Ultra-low-power UWB for sensor network applications. In *Circuits and Systems, 2008. ISCAS 2008. IEEE International Symposium on*, pages 2562–2565, May 2008.

[150] Eytan Modiano, Devavrat Shah, and Gil Zussman. Maximizing Throughput in Wireless Networks via Gossiping. In *Proc. of the SIGMETRICS Conf.*, pages 27–38, June 2006.

[151] David Moss and Philip Levis. BoX-MACs: Exploiting Physical and Link Layer Boundaries in LowPower Networking. Technical report, 2008.

[152] Luca Mottola, Gian Pietro Picco, and Adil Amjad Sheikh. FiGaRo: Fine-Grained Software Reconfiguration in Wireless Sensor Networks. In *Proc. of the EWSN Conf.*, pages 286–304, January 2008.

[153] Razvan Musaloiu-E., Chieh-Jan Mike Liang, and Andreas Terzis. Koala: Ultra-Low Power Data Retrieval in Wireless Sensor Networks. In *Proc. of the IPSN Conf.*, pages 421–432, April 2008.

[154] National Renewable Energy Laboratory NREL. Measurement and Instrumentation Data Center. [Online] `http://www.nrel.gov/midc/hsu/`, August 2013.

[155] Fredrik Österlind and Adam Dunkels. Approaching the Maximum 802.15.4 Multi-hop Throughput. In *Proc. of the HotEmNets*, June 2008.

[156] Fredrik Österlind, Adam Dunkels, Joakim Eriksson, Niclas Finne, and Thiemo Voigt. Cross-Level Sensor Network Simulation with COOJA. In *Proc. of the LCN Conf.*, pages 641–648, November 2006.

[157] Jeongyeup Paek, Ben Greenstein, Omprakash Gnawali, Ki-Young Jang, August Joki, Marcos Vieira, John Hicks, Deborah Estrin, Ramesh Govindan, and Eddie Kohler. The Tenet Architecture for Tiered Sensor Networks. *ACM Transactions on Sensor Networks*, 6(4):1–44, 2010.

[158] Jeongyeup Paek, JeongGil Ko, Jongsoo Jeong, Jongjun Park, Jong Jun, and Omprakash Gnawali. DualMOP-RPL: Supporting Multiple Modes of Downward Routing in a Single RPL Network. *ACM Transactions on Sensor Networks (TOSN)*, 11(2), May 2015.

[159] Seungmin Park, Jin Won Kim, Kwangyong Lee, Kee-Young Shin, and Daeyoung Kim. Embedded sensor networked operating system. In *Proc. of the IEEE ISORC Sump.*, pages 1–5, April 2006.

[160] Alessandro Pinto. Methods and Tools to Enable the Design and Verification of Intelligent Systems. *AIAA Infotech at Aerospace*, 2012.

[161] Kristofer S.J. Pister. Smart Dust. [Online] `http://robotics.eecs.berkeley.edu/~pister/SmartDust/SmartDustBAA97-43-Abstract.pdf`, 1997.

[162] Joseph Polastre, Jason Hill, and David Culler. Versatile Low Power Media Access for Wireless Sensor Networks. In *Proc. of the ACM SenSys Conf.*, pages 95–107, November 2004.

[163] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *Proc. of the IPSN Conf.*, pages 364–369, April 2005.

[164] Daniele Puccinelli, Silvia Giordano, Marco Zuniga, and Pedro José Marrón. Broadcast-free Collection Protocol. In *Proc. of the ACM SenSys Conf.*, pages 29–42, November 2012.

[165] Ragunathan Rajkumar, Insup Lee, Lui Sha, and John Stankovic. Cyber-physical systems: the next computing revolution. In *Proc. of the Design Automation Conf.*, pages 731–736, June 2010.

[166] Janos Sallai, Akos Ledeczi, and Peter Volgyesi. Acoustic Shooter Localization with Minimal Number of Single-Channel Wireless Sensor Nodes. In *Proc. of the ACM SenSys Conf.*, pages 96–107, November 2011.

[167] Thomas Schmid, Prabal Dutta, and Mani B. Srivastava. High-resolution, Low-power Time Synchronization an Oxymoron No More. In *Proc. of the IPSN Conf.*, pages 151–161, April 2010.

[168] Mingoo Seok, Dongsuk Jeon, C. Chakrabarti, D. Blaauw, and D. Sylvester. A 0.27V 30MHz 17.7nJ/transform 1024-pt complex FFT core with super-pipelining. In *Proc. of the IEEE ISSCC Conf.*, pages 342–344, feb 2011.

[169] Alireza Seyedi and Biplab Sikdar. Modeling and analysis of energy harvesting nodes in wireless sensor networks. In *Proc. of Communication, Control, and Computing Conf.*, pages 67–71, September 2008.

[170] Mo Sha, Gregory Hackmann, and Chenyang Lu. Energy-efficient Low Power Listening for Wireless Sensor Networks in Noisy Environments. In *Proc. of the IPSN Conf.*, pages 277–288, April 2013.

[171] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). Internet Engineering Task Force (IETF), Request for Comments (RFC): 7252, June 2014.

[172] Victor Shnayder, Mark Hempstead, Bor rong Chen, Geoff Werner Allen, and Matt Welsh. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the ACM SenSys Conf.*, pages 188–200, November 2004.

[173] Sigma Designs. Z-Wave. [Online] `http://www.z-wave.com/`, 2015.

[174] Gyula Simon, Miklós Maróti, Ákos Lédeczi, György Balogh, Branislav Kusy, András Nádas, Gábor Pap, János Sallai, and Ken Frampton. Sensor Network-Based Countersniper System. In *Proc. of the ACM SenSys Conf.*, pages 1–12, November 2014.

[175] Kannan Srinivasan, Prabal Dutta, Arsalan Tavakoli, and Philip Levis. An empirical study of low-power wireless. *ACM Trans. Sen. Netw.*, 6(2):16:1–16:49, March 2010.

[176] Kannan Srinivasan, Maria A. Kazandjieva, Saatvik Agarwal, and Philip Levis. The β-factor: Measuring Wireless Link Burstiness. In *Proc. of the ACM SenSys Conf.*, pages 29–42, November 2008.

[177] Vijay Srinivasan, John Stankovic, and Kamin Whitehouse. FixtureFinder: Discovering the Existence of Electrical and Water Fixtures. In *Proc. of the IPSN Conf.*, pages 115–128, April 2013.

[178] Gerald Stanje, Paul Miller, Jianxun Zhu, Alexander Smith, Olivia Winn, Robert Margolies, Maria Gorlatova, John Sarik, Marcin Szczodrak, Baradwaj Vigraham, Luca Carloni, Peter Kinget, Ioannis Kymissis, and Gil Zussman. Demo: Organic solar cell-equipped energy harvesting active networked tag (EnHANT) prototypes. In *Proc. of the ACM SenSys Conf.*, pages 385–386, November 2011.

[179] John A. Stankovic, Insup Lee, Aloysius Mok, and Raj Rajkumar. Opportunities and Obligations for Physical Computing Systems. *Computer*, 38:23–31, November 2005.

[180] Varun Subramanian, Michael Gilberti, Alex Doboli, and Dan Pescaru. A Goal-oriented Programming Framework for Grid Sensor Networks with Reconfigurable Embedded Nodes. *ACM Trans. Embed. Comput. Syst.*, 11(4):79:1–79:30, January 2013.

[181] Marcin Szczodrak and Luca Carloni. Demo: A complete framework for programming event-driven, self-reconfigurable low power wireless networks. In *Proc. of the ACM SenSys Conf.*, pages 415–416, November 2011.

[182] Marcin Szczodrak, Omprakash Gnawali, and Luca P. Carloni. Dynamic Reconfiguration of Wireless Sensor Networks to Support Heterogeneous Applications. In *Proc. of IEEE DCOSS Conf.*, pages 52–61, May 2013.

[183] Marcin Szczodrak, Omprakash Gnawali, and Luca P. Carloni. Modeling and Implementation of Energy Neutral Sensing Systems. In *Proc. of ENSSys Work.*, pages 9:1–9:6, November 2013.

[184] Marcin Szczodrak, Yong Yang, Dave Cavalcanti, and Luca P. Carloni. An Open Framework to Deploy Heterogeneous Wireless Testbed for Cyber-Physical Systems. In *Proc. of the IEEE SIES Symp.*, pages 215–224, June 2013.

[185] Robert Szewczyk, Eric Osterweil, Joseph Polastre, Michael Hamilton, Alan Mainwaring, and Deborah Estrin. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, June 2004.

[186] Janos Sztipanovits, Xenofon Koutsoukos, Gabor Karsai, Nicholas Kottenstette, Panos Antsaklis, Vijay Gupta, Bill Goodwine, John Baras, and Shige Wang. Toward a Science of Cyber-Physical System Integration. *Proc. of the IEEE*, 100(1):29–44, 2012.

[187] Arsalan Tavakoli, Aman Kansal, and Suman Nath. On-line sensing task optimization for shared sensors. In *Proc. of the IPSN Conf.*, pages 47–57, April 2010.

[188] Texas Instruments. 2.4 GHz IEEE 802.15.4 / ZigBee-ready RF Transceiver. [Online] `http://www.ti.com/lit/ds/symlink/cc2420.pdf`, 2013.

[189] Texas Instruments. MSP430F1611:16-bit Ultra-Low-Power MCU, 48kB Flash, 10240B RAM, 12-Bit ADC, Dual DAC, 2 USART, I2C, HW Mult, DMA. [Online] `http://www.ti.com/product/msp430f1611`, 2014.

[190] Thread Group. Thread: Powerful Technology Designed for the Home. [Online] `http://www.threadgroup.org/Technology.aspx`, 2015.

[191] S. Tin and A. Lal. SAW-based radioisotope-powered wireless RFID/RF transponder. In *Ultrasonics Symposium*, pages 1498–1501, October 2010.

[192] Gilman Tolle and David Culler. Design of an Application-Cooperative Management System for Wireless Sensor Networks. In *Proc. of EWSN Work.*, pages 121–132, February 2005.

[193] Unicomp. UCMote Proton B. [Online] `http://www.ucmote.com/en/products/1/24/ucmote-proton-b`, December 2014.

[194] Jean-Philippe Vasseur and Adam Dunkels. *Interconnecting Smart Objects with IP: The Next Internet*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2010.

[195] B. Vigraham and P.R. Kinget. An ultra low power, compact UWB receiver with automatic threshold recovery in 65 nm CMOS. In *IEEE RFIC Symp.*, volume jun, pages 251–254, 2012.

[196] Yin Wang, Yuan He, Xufei Mao, Yunhao Liu, and Xiang-Yang Li. Exploiting Constructive Interference for Scalable Flooding in Wireless Networks. *IEEE/ACM Trans. Netw.*, 21(6):1880–1889, December 2013.

[197] Brett A. Warneke, Michael D. Scott, Brian S. Leibowitz, Lixia Zhou, Colby L. Bellew, J. Alex Chediakt, Joseph M. Kahn, Bernhard E. Boser, and Kristofer S.J. Pister. An autonomous 16 mm^3 solar-powered node for distributed wireless sensor networks. In *Proc. of IEEE Sensors*, volume 2, pages 1510–1515, 2002.

[198] Alex S. Weddell, Michele Magno, Geoff V. Merrett, Davide Brunelli, Bashir M. Al-Hashimi, and Luca Benini. A Survey of Multi-source Energy Harvesting Systems. In *Proc. of the DATE Conf.*, pages 905–908, March 2013.

[199] Geoffrey Werner-Allen, Patrick Swieskowski, and Matt Welsh. MoteLab: a wireless sensor network testbed. In *Proc. of the IPSN Conf.*, pages 483–488, April 2005.

[200] T. Winter, P. Thubert, A. Brandt, J. Hui, R. Kelsey, P. Levis, K. Pister, R. Struik, J P. Vasseur, and R. Alexander. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks. Internet Engineering Task Force (IETF), Request for Comments (RFC): 6550, March 2012.

[201] Dingming Wu, Chao Dong, Shaojie Tang, Haipeng Dai, and Guihai Chen. Fast and fine-grained counting and identification via constructive interference in wsns. In *Proc. of the IPSN Conf.*, pages 191–202, April 2014.

[202] Yin Wu and Wenbo Liu. Routing protocol based on genetic algorithm for energy harvesting-wireless sensor networks. *Wireless Sensor Systems, IET*, 3(2):112–118, July 2013.

[203] John Yackovich, Daniel Mosse, Anthony Rowe, and Raj Rajkumar. Making WSN TDMA Practical: Stealing Slots Up and Down the Tree. In *Proc. of the RTCSA Conf.*, pages 41–50, August 2011.

[204] F.M. Yaul and A.P. Chandrakasan. 11.3 A 10b 0.6nW SAR ADC with data-dependent energy savings using LSB-first successive approximation. In *Proc. of the IEEE ISSCC Conf.*, pages 198–199, February 2014.

[205] Lohit Yerva, Apoorva Bansal, Bradford Campbell, Prabal Dutta, and Thomas Schmid. An IEEE 802.15.4-compatible, Battery-free, Energy-harvesting Sensor Node. In *Proc. of the ACM SenSys Conf.*, pages 389–390, November 2011.

[206] Lohit Yerva, Brad Campbell, Apoorva Bansal, Thomas Schmid, and Prabal Dutta. Grafting energy-harvesting leaves onto the sensornet tree. In *Proc. of the IPSN Conf.*, pages 197–208, April 2012.

[207] Shengrong Yin, Omprakash Gnawali, Philipp Sommer, and Branislav Kusy. Multi Channel Performance of Dual Band Low Power Wireless Network. In *Proc. of the IEEE MASS*, pages 345–353, October 2014.

[208] Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting Concurrent Applications in Wireless Sensor Networks. In *Proc. of the ACM SenSys Conf.*, pages 139–152, November 2006.

[209] S. Zahedi, M. Szczodrak, Ping Ji, D. Mylaraswamy, M. Srivastava, and R. Young. Tiered architecture for on-line detection, isolation and repair of faults in wireless sensor networks. In *Proc. of the IEEE MILCOM Conf.*, pages 1–7, November 2008.

[210] Jin Zhang, Qian Zhang, Yuanpeng Wang, and Chen Qiu. A Real-time Auto-adjustable Smart Pillow System for Sleep Apnea Detection and Treatment. In *Proc. of the IPSN Conf.*, pages 179–190, April 2013.

[211] Feng Zhao and Leonidas Guibas. *Wireless Sensor Networks: An Information Processing Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[212] Xiaolong Zheng, Zhichao Cao, Jiliang Wang, Yuan He, and Yunhao Liu. ZiSense: Towards Interference Resilient Duty Cycling in Wireless Sensor Networks. In *Proc. of the ACM SenSys Conf.*, pages 119–133, November 2014.

[213] Jianxun Zhu, Gerald Stanje, Robert Margolies, Maria Gorlatova, John Sarik, Zainab Noorbhaiwala, Paul Miller, Marcin Szczodrak, Baradwaj Vigraham, Luca P. Carloni, Peter R. Kinget, Ioannis Kymissis, and Gil Zussman. Demo: prototyping UWB-enabled EnHANTS. In *Proc. of the MobiSys Conf.*, pages 387–388, June 2011.

[214] ZigBee Alliance. New ZigBee green power feature set revealed, June 2009.

[215] Marco Zimmerling, Federico Ferrari, Luca Mottola, Thiemo Voigt, and Lothar Thiele. pTunes: Runtime Parameter Adaptation for Low-power MAC Protocols. In *Proc. of the IPSN Conf.*, pages 173–184, April 2012.

[216] Hubert Zimmermann. OSI Reference Model - The ISO Model of Architecture for Open System Interconnection. In *IEEE Trans. Commun.*, volume 28, pages 425–432, April 1980.

[217] Zolertia. Z1 Platform. [Online] `http://zolertia.com/products/z1`.