Multitasking on Low-Power Wireless Distributed Embedded Systems

Thesis proposal

Marcin Krzysztof Szczodrak
Department of Computer Science
Columbia University
msz@cs.columbia.edu

January 18, 2014

Abstract

A Wireless Sensor Network (WSN) is a system of low-power tiny computers interconnected over a radio and sensing surrounding environment. Despite many applications and industry interest, WSNs are not yet commercialized. The WSN system dynamics imply design conditions that make programming difficult, thus raising doubts on the perspective of a successful application of this technology. The main application programming challenge is that one cannot express WSN data computation without deciding how the data is communicated. For my thesis I propose to study the relation between data computation and communication across the WSN in order to simplify application programming and maintenance while preserving system performance. This document describes the two contributions that I have made so far, Fennec Fox and Open Testbed Framework, and presents a plan for the completion of my dissertation.

The first contribution is Fennec Fox. Here I 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 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.

The second contribution is the Open Testbed Framework. Here I present a set of tools for deploying heterogeneous wireless testbeds for WSNs. The testbed architecture, which can be configured and optimized for each particular deployment, consists of: motes, a testbed 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 WSN application software and the supporting firmware. I show 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 present experiments normally conducted by WSN engineers to better understand the environment in which the WSN is deployed. The results of these experiments show the feasibility of the proposed framework in assisting WSN research and development.

To complete my thesis I plan to address the challenges of the application programming by introducing system processes providing computation services that hide the complexities of the WSN dynamics.

Contents

1 Introduction

The Wireless Sensor Network (WSN) is a technology to enable the engineers' vision of a world embedded with tiny, wireless, and self-sustainable devices interacting with the surrounding environment [4, 12, 52]. Researchers have envisioned a wide spectrum of applications areas for WSNs, including: construction [33, 44, 82], transit [5, 8, 51], energy [21, 24, 64], military [54, 76, 80], environment [35, 62, 88], and health-care [34, 46, 97]. Still, after over a decade of academic research, this technology fails to be successfully commercialized. One challenge is the difficulty of programming a network of microcontrollers [55] that are constrained in terms of energy, computation, communication and memory. Another challenge is the limited functionality of the WSN deployments: once successfully programmed and installed, each WSN runs with only one application.

Trends and Challenges. Research in WSN has focused on improving the quality of the embedded software and on managing the limited energy resources. Since 2004, TinyOS [58] and Contiki [15] have been the leading development frameworks for programming wireless mesh networks of embedded devices. Both frameworks consist of network libraries for data collection [26, 47] and dissemination [23, 60], with support of TCP/IP [14, 26, 37] and the necessary tool-chain for programing and installing embedded firmware. Researchers continuously improve the lifetime of the wireless devices by duty-cycling their radio [74, 84], using models of wireless radio communication [9], and applying radio link quality estimation [2, 81], signal interference [20, 23] and cancellation [40]. All these research efforts address the physical limitations of the low-power wireless embedded devices.

The difficulty of programming a network, that is translating the application logic into the software running across the mesh of nodes, is the major roadblock toward the commercialization of WSNs. The thread support added to TinyOS and Contiki did not simplify the task of event-driven software programming [17, 45, 55, 68]. The introduction of virtual machines and new programming languages for the existing embedded operating systems [29, 56, 65, 66, 72, 77] has not closed the gap between the applications' logic and the low-level software drivers for hardware components. An application firmware that is built together with specific communication mechanisms and primitive operating-system support is difficult to program and extremely complex to debug and maintain. On the other hand, hiding the complexity of the mesh network and the limitations of the physical resources from the application design results in inefficient system deployments because it exhausts energy resources and delays network traffic.

The research community continues to address the motes' low-level physical constraints, while on the software programming level it debates between two solutions. The first approach proposes to expose the mesh network complexity to preserve system performance. The second solution hides the communication problems at the price of some performance degradation. There is no consensus on the programming direction and on the definition of a WSN application process. I believe that the challenge lies in understanding the structure of a process executing on WSN and in answering the fundamental research question: *Are communication mechanisms part of the process logic?*

Thesis Proposal. For my thesis I propose to study the low-power wireless network design in order to simplify application programming and maintenance while preserving system performance. The proposed research seeks to: (1) decouple the network operation processes from the user application logic, (2) facilitate multiprocessing across a single wireless network, (3) introduce system

processes providing services to the application programs, (4) separate the computation logic from the communication constraints during application programming, (5) prototype a system showing feasibility of multiprocessing and decoupling system logic from application logic, and (6) define an application programming interface that separates the user space from the network system space.

To simplify the user application programming, I take a hybrid approach to WSN design: the system processes involve both the computation and the communication logic, but the application processes only express the computation logic. The system processes are exposed to the communication complexity of the WSN and therefore they can control the network performance. The application processes are not aware of the distributed system nature, so they process the data stored across the whole WSN by invoking system calls. The system calls express the computation logic that is executed by a sequence of the system processes running on the WSN.

In my thesis I show the feasibility of executing multiple processes on the WSN. At the current state-of-the-art every node in the network executes a single firmware consisting of an application code integrated with an operating system library. To go beyond the single functionality of the distributed network I define the notion of *network process*. In particular, I examine the relation between the computation aspects of the application processes executed by the WSN and the necessary communication requirements that allow these applications' work to be performed as expected. With the notion of network process, I concentrate on the software implementation and the run-time modeling of the network-process *context switch*, which enables multiprocessing across the wireless mesh of nodes.

Next, I define the difference between the system processes and the user application processes. This distinction between the two types of processes running on the network will introduce the concept of a distributed operating system to the WSN. At the current state-of-the-art, embedded operating systems are executing on each individual node of the network. Instead I research the primitives for building an operating system support across the whole network, one providing an abstraction of a single system. Therefore, I focus on: the relation between the system services and the user applications, how the system controls the application execution, and what services such distributed system should provide to user application processes.

Broader Impact. After completion of the thesis we will better understand the software engineering gap between the WSN applications and the available technological resources. The proposed separation of the system processes, consisting of the computation and communication details, from the application processes, which are not exposed to WSN communication complexity, will serve as a benchmark for future works. In the long-term, this thesis lays theoretical groundwork, supported with empirical results, for future low-power wireless embedded systems. This research aims to make this technology available to the application domain experts. From this will emerge new applications for improving human health-care, climate monitoring, social communication, game industry and media.

Research Contributions. The following tasks have been completed toward the thesis. First, by developing the Fennec Fox framework I addressed the problem of scheduling and supporting the execution of multiple heterogeneous applications on top of the same WSN [85, 86]. I verified the hypothesis that communication protocols supporting one application are not suitable to enable the execution of another application with different performance constraints. To answer this challenge, Fennec Fox introduced the mechanisms to dynamically reconfigure the protocol stack in order to adapt the communication services to the application requirements. The Fennec Fox framework is discussed in Chapter 3.

Second, by developing the Open Testbed Framework, I provided a comprehensive set of tools for deploying, testing, reconfiguring, and evaluating the WSN application software and supporting firmware [87]. Although the remote testbeds enable the evaluation of a system prototype against different wireless network topologies and deployment scenarios, they are limited in understanding and processing signals from the actual interaction with the physical world because WSN developers do not have access to the target environment. I solved this problem with tools that enable rapid and low-cost setup of a local testbed. Chapter 4 presents more details about the Open Testbed Framework. In the following chapter I compare related work against this thesis proposal. Finally, in Chapter 5 I present my plan to complete the thesis.

2 Related work

2.1 Network Communication Reconfiguration

A major approach to WSN reconfiguration is based on distributing fragments of code that are loaded and executed on the sensor nodes. TinyCubes [67], Enix [10], BASE [31], and ViRe [6] are examples of such systems. Works such as Deluge can perform full-program update on the nodes [36]. Some systems allow users to program WSNs at run-time. Maté [56] allows dissemination of code to be executed in a virtual machine. Tenet [72] allows sending data-flow programs to be executed in the network. My 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 [69]), using a shared infrastructure [90], or minimizing redundant broadcast transmissions as done by Trickle [59]. 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 WSN stacks such as Rime [16], uIP [14], and TinyOS IP stack [38] that try to follow the layered protocol model. Besides these complete stacks, there are now layer-specific protocols (e.g., CTP [26], XMAC [7]) that are designed to allow different protocols at the higher or lower layers of the stack. With Fennec Fox, I leverage the design and implementation of these protocols to provide a framework that allows applications to compose their own stack using protocols of their choice at each layer.

MultiMAC [92] shows that different MACs can be implemented on top of the same radio driver. The pTunes project [98] 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.

2.2 Wireless Sensor Network Testing and Deployment

Over the years two simulators have gained popularity in the WSN research community: TOSSIM [57] and COOJA [71]. With TOSSIM, it is possible to simulate the execution of software applications written in the nesC language [25] running on a network of motes on top of the TinyOS operating

system environment [58] 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 operating system [15] 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 WSN applications and the design of wireless infrastructure to support them cannot prescind from the use of testbeds.

Motelab was one of the first successful WSN testbeds [94]. 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 WSN 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 [3]. 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 [74], 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 [32]. 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 [13]. With a backbone infrastructure consisting of 6 Mac Mini PCs and a network of USB hubs and cables, Indriya is geographically the largest WSN 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 WSNs. 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. WSN researchers and engineers, however, cannot completely evaluate their work on these testbeds because WSN applications require continuous interaction with the physical world. Therefore, during the WSN instrumentation one must have access to the target environment of deployment as well as 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. The Open Testbed Framework complements the existing WSN testbeds because it enables to understand the data that are processed by a WSN application. My framework helps engineers and researchers in setting up their own local testbeds supporting the development of WSN.

3 Dynamic Reconfiguration of Wireless Sensor Networks to Support Heterogeneous Applications

The following is an excerpt from the work on Fennec Fox that was published with Omprakash Gnawali from the University of Houston and my advisor Luca P. Carloni [86].

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 [72].
- Re-program the WSN using systems such as Deluge [36] when we switch from *Collection* to *Firecam*.
- Re-configure the MAC parameters with systems such as pTunes [98] 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 paper) 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 extendible to a large number of applications and various protocols.

3.2 One Network, Two Applications

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

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

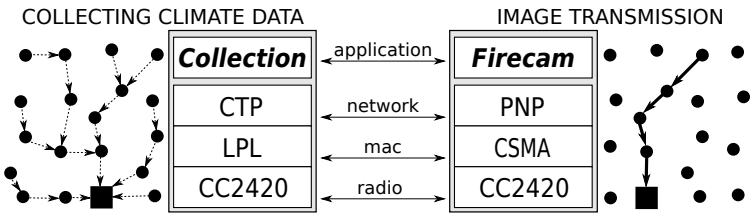


Figure 1: Two different applications and their corresponding protocol stacks. In the sequel I use the term *CTP stack* and *PNP stack* to denote the protocol stacks required by *Collection* and *Firecam*, respectively.

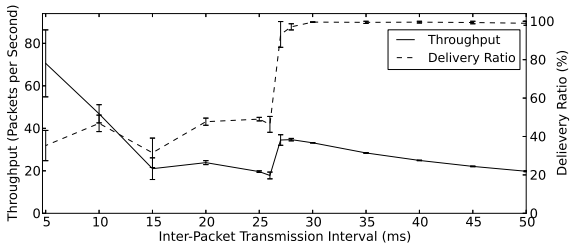
series of pictures and sends this stream of pictures to a sink node, which may or may not coincide with the same sink node of the other application. Thus, differently from the previous application, in the case of *Firecam* we are interested in the transmission of large amounts of data (the size of a 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].

Fig. 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 [26] 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.

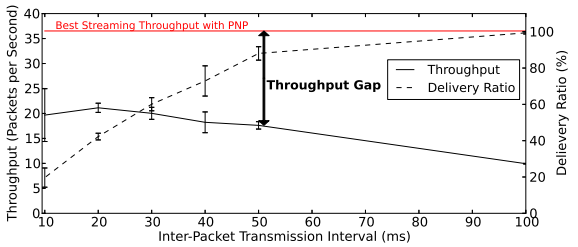
In the case of *Firecam*, instead, the efficient transmission of a stream of pictures from one particular node to the sink requires to quickly establish a multi-hop path between them. Assuming that a picture size is 240×320 pixels and that each pixel is encoded as a single byte, the transmission of a 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. [43] and Österlind et al. [70], 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 Fig. 1: *each of them supports well the corresponding application, for*

[2]This is indeed a practical issue since many emergency alarms are often the results of false positive sensor readings.



(a) Throughput and Delivery Ratio during operation of the PNP stack.



(b) CTP with CSMA does not support high point-to-point throughput.

Figure 2: For point-to-point communication PNP is a better network protocol than CTP.

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 [13], which consists of 119 active TelosB motes [74]. 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 4.4.) 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*. Fig. 2(a) 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 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. Fig. 2(b) 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

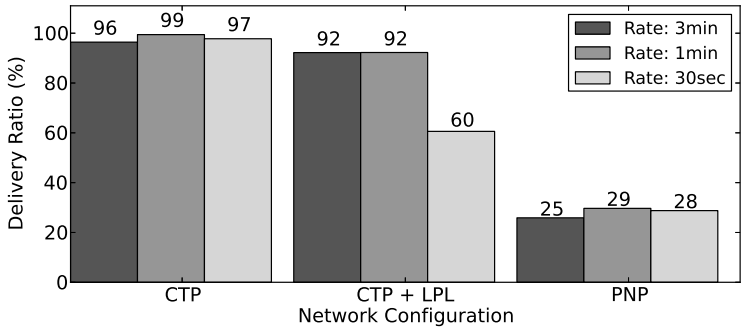


Figure 3: PNP cannot support the same many-to-one delivery ratio as CTP.

88% and 60% for lower inter-packet transmission intervals equal to 50ms and 30ms, respectively. As highlighted in Fig. 2(b), there is a clear throughput gap between the performance of the two protocol stacks of Fig. 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 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 Fig. 2(a) and Fig. 2(b), 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*. Fig. 3 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 applications 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]. On the other hand, we are focusing on a heterogeneous application scenario that does not require simultaneous execution of the

[3]pTunes [98] only allows to reconfigure MAC parameters and Deluge [36] can change the MAC by reprogramming the whole sensor node firmware.

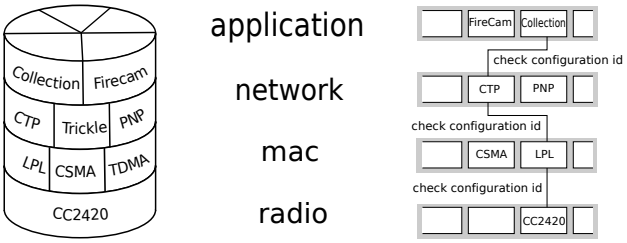


Figure 4: The Fennec Fox four-layer protocol stack.

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 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. Fig. 4 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 [25] on top of the TinyOS operating system [58]. 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.

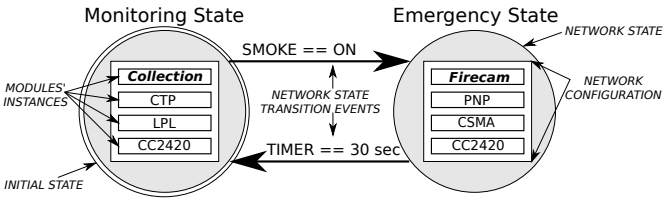
The network protocol stack is implemented as a set of switch statements, which direct function calls and transfer packets among the modules based on the configuration id, as shown in Fig. 4. The id determines every function call made outside of the module's layer. To allow a radio driver to dispatch 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 [39]. The value of the packet's id is set to the id of the configuration of the stack in which that packet was created.

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, Fig. 5(a) 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 Fig. 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

[4] In this paper, 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.



(a) A FSM model of a WSN supporting the *Collection* and *Firecam*.

```
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
```

(b) Swift Fox program reconfiguring WSN between two applications.

Figure 5: Fennec Fox Framework model of execution and its programming language.

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, Fig. 5(b) 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 Fig. 5(a) 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* 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 *Collec-*

tion 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 [59] algorithm. First, no messages are disseminated when network reconfiguration does not take place and all nodes in the network run the same configuration. 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

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
```

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
```

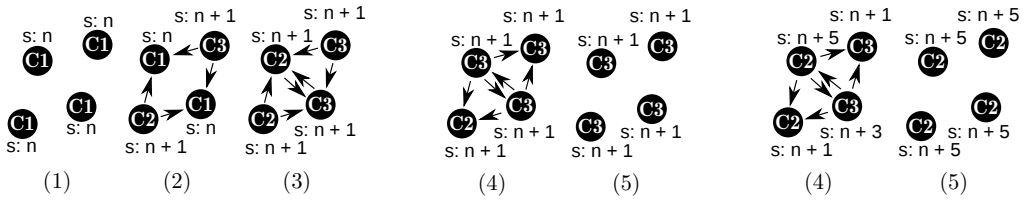
ms (lines 5-7)[5]. The BCP terminates after r broadcasts attempts (lines 1-2, 8-9).

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 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

[5]This transmission suppression avoids unnecessary radio broadcasts, in a way similar to the Trickle protocol [59].

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



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

Figure 6: Network synchronization: (b) deterministic and (c) non-deterministic.

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 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 Fig. 6(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 Fig. 6(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 Fig. 6(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 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.

[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.

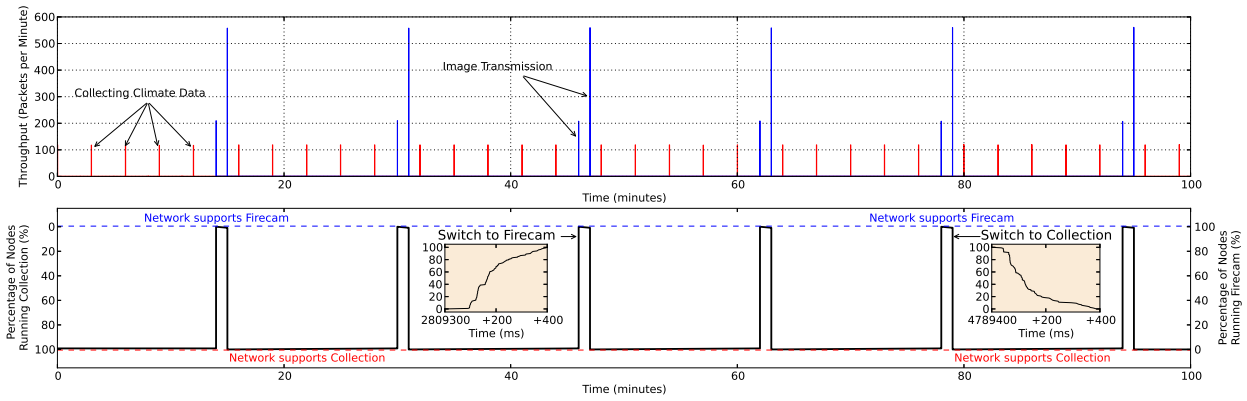


Figure 7: A 100 minute run of a network reconfiguring between the $Collection$ and $Firecam$ applications.

3.4 Sample 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.

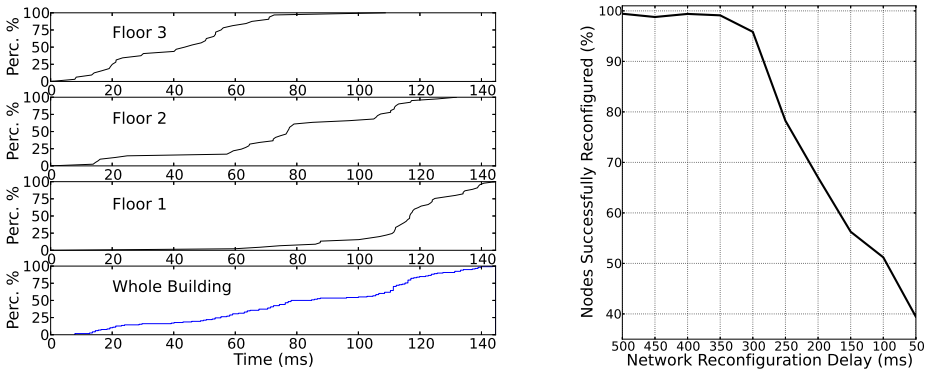
Code and Memory Overhead. On TelosB, the reconfiguration protocols and mechanisms introduce an overhead of 4.7 KB of ROM and 0.2 KB of RAM. The Swift Fox program with both *Collection* and *Firecam* requires 28.9 KB of ROM and 5.6 KB of RAM.

Single-node Reconfiguration Delay. 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



(a) Building Reconfiguration (b) Transition Rate

Figure 8: Reconfiguration performance with radio duty-cycled.

minute, the network is reconfigured back to run *Collection* and the process is repeated. The lower graph in Fig. 7 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 Fig. 7. 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 Fig. 7. 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 Fig. 7 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 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. Fig. 8(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. Fig. 8(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 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. We ran an 8-hour experiment with network firing reconfiguration event every 500ms. On average, 99.91% of network reconfigurations were 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 followed each network configuration, sending 0.4 messages per configuration transition.

3.5 Summary

We studied 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.

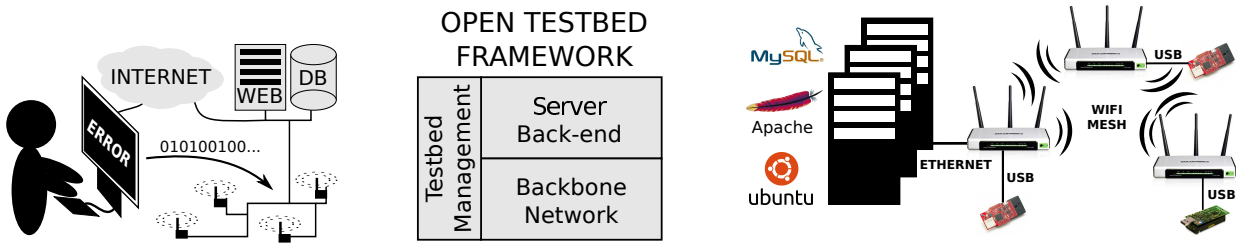
4 An Open Framework to Deploy Heterogeneous Wireless Testbeds for Cyber-Physical Systems

The following is an excerpt from the work on Open Testbed Framework that was published with Yong Yang and Dave Cavalcanti from the Philips Research North America and my advisor Luca P. Carloni [87].

4.1 Introduction

Many cyber-physical system (CPS) [75] 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 [42, 74, 99] and minimal operating system support [15, 22, 58]. The interaction of the cyber infrastructure with the physical world is controlled by a distributed, concurrent, and heterogeneous system [12, 83]. The design of such system and the programming of the application software is a complex engineering task [53, 89], which includes a critical validation step [48, 73] that requires physical implementation.

During new system installation, CPS engineers and researchers can find only limited help in the use of network simulators [57, 71] and remote network testbeds [3, 11, 13, 32, 61, 94]. 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.



(a) Testbed Control (b) Testbed Framework (c) Testbed Framework Architecture

Figure 9: (a) An engineer tests a program on a remote testbed. (b) High-level structure overview of the framework . (c) Actual testbed implementation.

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 Fig. 9(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 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 Fig. 9(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 paper is organized as follows. Section 2.2 discusses related work. Section 4.2 describes the framework's components, while Section 4.3 presents the two case studies. Section 4.4 provides sample testbed statistical information for early stage CPS prototyping.

4.2 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.

Fig. 9(c) 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.

4.3 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.

4.3.1 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.

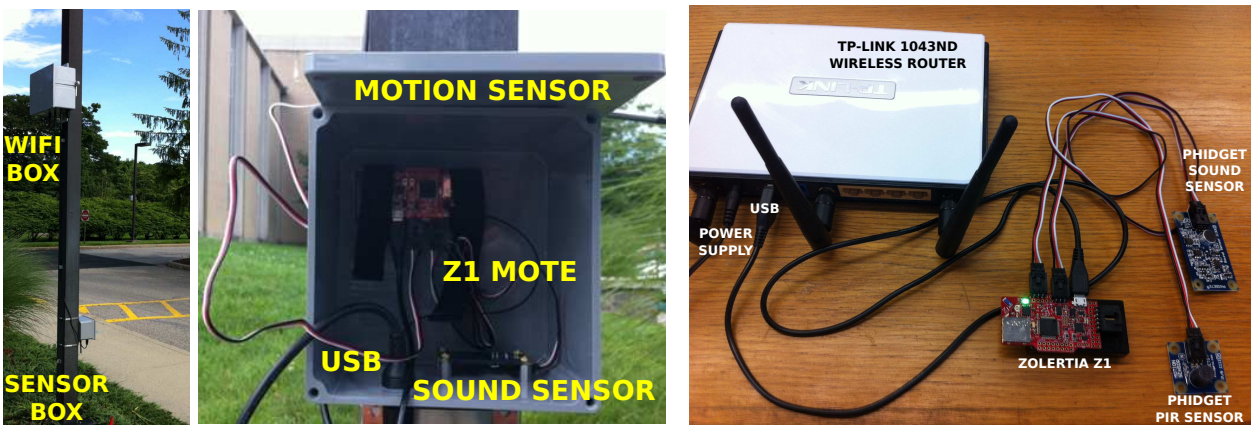
Hardware Infrastructure. Fig. 10(a) shows the mounting of the the testbed hardware on one of the light poles. Each pole comprises of one WiFi box and one sensor box. Each WiFi box contains a TP-LINK 1043ND WiFi router with a 400MHz microcontroller and 8MB of Flash and 32MB of RAM memories. The router 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.

Fig. 10(b) shows an opened sensor box. The box contains a Zolertia Z1 [99] 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.

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, 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



(a) Light Pole (b) Inside Sensor Box (c) Indoor Testbed Node

Figure 10: (a-b) Testbed deployment on a parking lot. (c) Node with router, mote and sensors.

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.

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

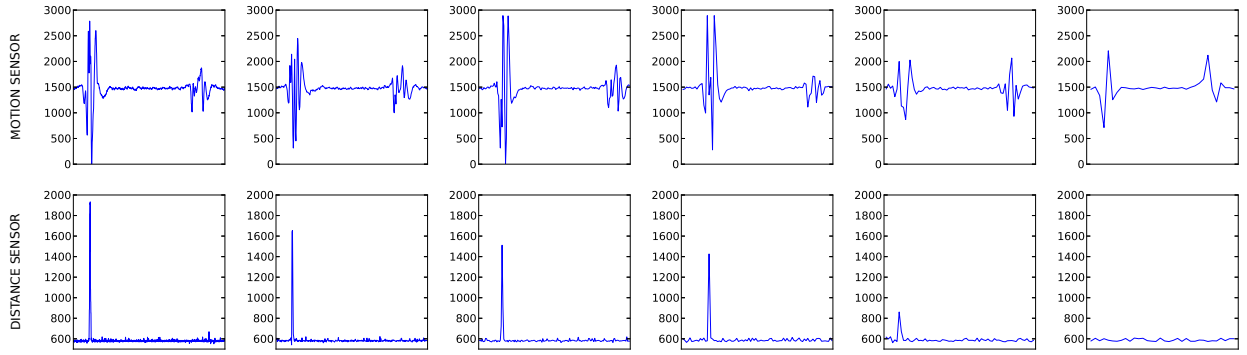
Fig. 10(c) shows one of the deployed testbed nodes: the TP-LINK 1043ND WiFi router is connected to the power source and to the Z1 mote through a USB cable. The Z1 mote is connected to two sensors, PIR (motion detection), through the available Phidget ports. Combined, the assembly of one testbed node 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 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 Fig. 10(c), 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 [86] and TinyOS [58]. 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.



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

Figure 11: 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.

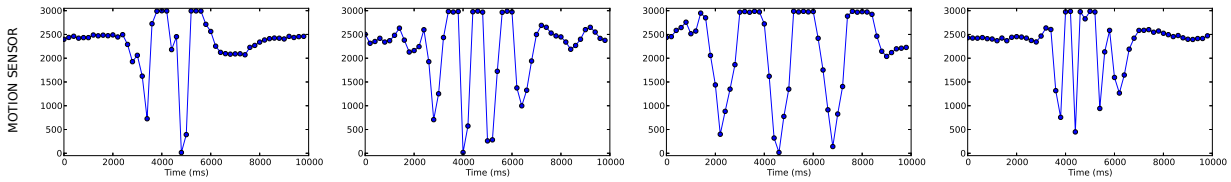
After studying the environment, in the second step, we ran the same embedded program as in the first step, but this time we used LPWN's multi-hop communication, instead of sending messages over the USB and WiFi routers. Recompiling the firmware to use LPWN instead of the USB connection and installing 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 [86] program that configures the embedded firmware.

4.4 Sample Testbed Evaluation

In this section we present an evaluation of the two testbed deployments introduced in Section 4.3, and on these examples we show how to implement a testbed for CPS prototyping. 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.

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 [64]. In another work, multiple distance sensors were used to track people walking between the rooms of a house [34]. 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



(a) Office Doorway (b) Parking (person 6ft) (c) Parking (person 12ft) (d) Parking (car)

Figure 12: 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.

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 [34]. In our experiments, we started with sampling every 25ms because that is the highest rate at which 99.76% of packets are successfully delivered [8]. 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.

Fig. 11 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 Fig. 11 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.

Fig. 12 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 mo-

[8]These experiments are in the longer paper version.

tion sensor measurements collected for a period of 10 seconds and with each single measurement marked as a dot. Fig. 12(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. Fig. 12(b) and Fig. 12(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. Fig. 12(d) shows a trace of measurements taken when a car was passing in front of the motion sensor.

As shown in Fig. 12, 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 Fig. 12(a) with the last one from Fig. 12(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 Fig. 11 and Fig. 12 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.

4.5 Summary

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.

5 Research plan

The following are the contributions I have made so far:

- I designed, implemented, and evaluated Fennec Fox framework and Swift Fox programming language to enable the execution of heterogeneous applications on a Wireless Sensor Network (WSN) [85, 86].
- I developed the Open Testbed Framework that provides necessary tools for rapid deployment and installation of a WSN testbed [87].

For the rest of my dissertation I propose to address the following problems:

- What are the WSN system processes and how they interact with applications (5.1).
- What is the set of system calls that must be exposed to the user applications and how these system services are executed on the WSN (5.2).

The details of the proposed work are discussed in the following sections.

5.1 System Processes

To understand the relation between system processes and applications, I will define, design, implement and evaluate examples of system utilities that are separated from the application logic, but are necessary for successful WSN operation. The examples of important WSN services are energy-management, resource discovery, and quality of information (QoI). This is ongoing work with some preliminary results in designing the energy-management system processes.

The WSN energy-management is one of the most important system processes because its actions define when and for how long the WSN is operational. Recent energy-harvesting improvements offer a spectrum of solutions to the problem of battery-constrained WSN life-time [93]. For example, WSN motes can be powered with solar energy [1, 18, 30] or by people's movement [27]. Motes can also communicate by reflecting TV signals [63] or by harvesting energy from a tiny radioisotope [91]. I will apply some of these techniques to enable the execution of sensing applications, which currently operate on non-rechargeable batteries. Modeling energy harvesting and consumption plays a critical role in the design of energy-neutral systems. Real-life energy-harvesting traces [49] and power consumption measurements [79] enable the creation of energy models. These models can be used to predict energy-harvesting rates [28] and to run system simulations [78]. I will introduce a feedback control model combining both the energy-harvesting and the energy-management algorithms. Energy-neutral systems require careful operation that is constantly aware of its energy resources. The network protocols use energy aware routing [59, 95] and new MAC protocols offer energy-conservation primitives [19, 96]. There are new application development methods for power-efficient sensing [41] and actuating [51]. I will show that designing an energy-neutral system should address both the system computation and communication

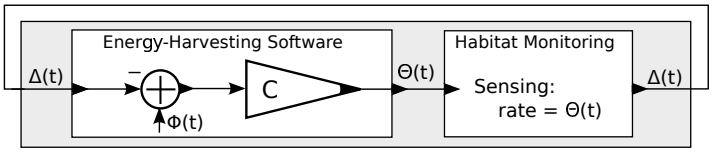


Figure 13: Feedback control model of a WSN system where application sensing rate is adjusted to the rate at which energy is harvested.

costs. The energy-management system processes are responsible for carefully spending the limited battery energy resources and for deciding how to utilize the harvested energy. I have already completed some preliminary work on modeling, implementation, and evaluation of a single WSN executing energy-harvesting algorithms and sensing applications. The network energy-management is modeled as a feedback control system.

Fig. 13 shows an example of a WSN model running with the energy-harvesting system processes and an application process that is collecting environmental sensor measurements. In this model, the application's sensing rate is controlled by the system process monitoring the WSN energy-resources. The application sensing rate is computed as a difference between the rate at which energy is consumed during application's execution - $\Delta(t)$, and the rate at which energy is harvested - $\Phi(t)$, multiplied by a scaling parameter C.

The two other system processes that I will design, implement, and demonstrate as part of the WSN system are: I/O resource discovery and quality of information of the sensor data. Resource discovery is a system process that provides information about the available inputs and outputs of the WSN, which at run-time can join and leave the network. The resource discover process will provide information about sensors and actuators as well as communication interfaces with other systems, such as UART or Ethernet. Quality of information is another important process, which quantifies the trust in the sensor measurements. Because sensor data impact the application's computation logic and the WSN system operation itself, understanding the quality of the information gathered from the sensors' measurements is critical for successful operation.

The examples of the three different WSN system services will help me to understand the relation between the system processes and the application processes. Successful implementation and evaluation of the energy-management, resource discovery and quality of information processes will provide additional evidence for the need of multiprocessing on the WSN. These services are important for robust and long-term deployments. Therefore, by prototyping them, I will expose the challenges that the industry is facing for commercializing the WSN.

5.2 Control Abstraction

This research direction focuses on system processes to simplify WSN application programming. These processes provide control abstraction, which enables to implement the application logic by invoking system calls. I propose to use two examples of system function calls that when invoked by the application logic execute a mathematical computation over the sensor measurements. The proposed computation methods are iterative algorithms for generating a sequence of improving approximate solutions, and methods for computing statistical relation between random variables.

Iterative algorithms are popular solutions to approximate variables of the WSN system. Using Jacobi Iteration as an exemplary algorithm, I propose to study the computation and communica-

tion tradeoffs of different implementations of this algorithm. The first implementation follows the central computation model, where a single node aggregates all the data and computes approximations. The second, alternative implementation adopts the free wireless message broadcast to enable parallel transmissions of data to all the nodes within a neighborhood. Applying wireless broadcast can accelerate data exchange, which in consequence can decrease the computation time.

Applications that are using in-network processing often compute statistical information about the sensor data. The related works in statistical computation in WSN either provide simple computations methods, such as computing the maximum value [50], or ignore the network communication overhead [65]. I propose to use examples of statistical methods, such as the maximum value, regression, and data correlation, to find their communication overheads, and to find examples of function calls that can become part of the interface to the application programming space. I will use these examples to establish a systematic methodology of adding more computational services, including functions that are compositions of simpler, statistical computation primitives.

The examples of the WSN processes providing mathematical computation services will allow me to conclude my thesis with the definition of a computation model for WSN application programming. I plan to prototype the WSN application programming interface, separating the user application space from the system processes domain.

5.3 Research Timeline

Wireless sensor networks are complex systems, very different from a single isolated computer or from existing distributed systems that are not exposed to so many physical constraints. Therefore, I propose to provide theoretical foundations for low-power wireless network design and evaluate them with a system prototype. I will show that by enabling multiprocessing across wireless sensor network and by adapting the wireless network communication to the computation demands, we can simplify application programming and address the challenges of realizing industrial WSN systems.

Table 1 shows my timeline for completion of the proposed research work.

Timeline	Work	Progress
Jan. 2014	Development of system services with Fennec Fox	ongoing
Mar. 2014	Finish implementation of the system services - Sec. 5.1	planned
Apr. 2014	Evaluate and Document the system services	planned
Jun. 2014	Finish implementation of the mathematical functions in Fennec Fox - Sec. 5.2	planned
Jul. 2014	Evaluate and Document distributed computation in WSN	planned
Oct. 2014	Dissertation Writing	planned
Dec. 2014	Dissertation Defense	planned

Table 1: Timeline for completion of my research.

Thus, I plan to defend my thesis in December 2014.

References

[1] Y. Afsar et al. Evaluating photovoltaic performance indoors. In *Proc. of the Photovoltaic Specialists Conf.*, pages 1948–1951, June 2012.

[2] Muhammad Hamad Alizai et al. Bursty traffic over bursty links. In *Proc. of the ACM SenSys Conf.*, pages 71–84, November 2009.

[3] Anish Arora et al. Kansei: A high-fidelity sensing testbed. In *Internet Computing, IEEE*, volume 10, pages 35–47, March 2006.

[4] Kevin Ashton. That 'internet of things' thing. 2009.

[5] Ravneet Bajwa et al. In-pavement wireless weigh-in-motion. In *Proc. of the IPSN Conf.*, pages 103–114, April 2013.

[6] Rahul Balani et al. Vire: Virtual reconfiguration framework for embedded processing in distributed image sensors. In *APRES Work.*, April 2008.

[7] Michael Buettner et al. 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.

[8] M. Ceriotti et al. 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.

[9] Alberto Cerpa et al. 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.

[10] 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.

[11] Geoff Coulson et al. Flexible experimentation in wireless sensor networks. *Commun. ACM*, 55(1):82–90, January 2012.

[12] David Culler, Deborah Estrin, and Mani Srivastava. Guest editors' introduction: Overview of sensor networks. *Computer*, 37:41–49, August 2004.

[13] 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.

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

[15] Adam Dunkels et al. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *Proc. of the IEEE LCN Conf.*, pages 455–462, November 2004.

[16] 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.

[17] 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.

[18] Prabal Dutta et al. Trio: enabling sustainable and scalable outdoor wireless sensor network deployments. In *Proc. of the IPSN Conf.*, pages 407–415, April 2006.

[19] Prabal Dutta et al. A building block approach to sensornet systems. In *Proc. of the ACM SenSys Conf.*, pages 267–280, November 2008.

[20] Prabal Dutta et al. Wireless ACK collisions not considered harmful. In *ACM HotNets Work.*, October 2008.

[21] 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.

[22] 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.

[23] F. Ferrari et al. Efficient network flooding and time synchronization with Glossy. In *Proc. of the IPSN Conf.*, pages 73–84, April 2011.

[24] Romain Fontugne et al. Strip, bind, and search: A method for identifying abnormal energy consumption in buildings. In *Proc. of the IPSN Conf.*, pages 129–140, April 2013.

[25] David Gay et al. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the PLDI*, pages 1–11, May 2003.

[26] Omprakash Gnawali et al. Collection tree protocol. In *Proc. of the ACM SenSys Conf.*, pages 1–14, November 2009.

[27] Maria Gorlatova et al. Movers and shakers: Kinetic energy harvesting for the internet of things. *CoRR*, abs/1307.0044, 2013.

[28] 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.

[29] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Macro-programming wireless sensor networks using kairos. In *Proc. of the DCOSS Conf.*, pages 126–140, June 2005.

[30] 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.

[31] Marcus Handte et al. The BASE plug-in architecture - composable communication support for pervasive systems. In *Proc. of the ICPS Conf.*, page 443, July 2010.

[32] Vlado Handziski et al. Twist: a scalable and reconfigurable testbed for wireless indoor experiments with sensor networks. In *Proc. of Work. REALMAN*, pages 63–70, May 2006.

[33] T. Harms, S. Sedigh, and F. Bastianini. Structural health monitoring of bridges using wireless sensor networks. *Instrumentation Measurement Magazine, IEEE*, 13(6):14–18, December 2010.

[34] Timothy W. Hnat et al. Doorjamb: unobtrusive room-level tracking of people in homes using doorway sensors. In *Proc. of the ACM SenSys Conf.*, pages 309–322, November 2012.

[35] Wen Hu et al. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *Proc. of the IPSN Conf.*, pages 503–508, April 2005.

[36] 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.

[37] Jonathan W. Hui and David Culler. Ip is dead, long live ip for wireless sensor networks. In *Proc. of the ACM SenSys Conf.*, pages 15–28, November 2008.

[38] J.W. Hui and D.E. Culler. IPv6 in low-power wireless networks. *Proc. of the IEEE*, 98(11):1865–1878, November 2010.

[39] 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`.

[40] Mayank Jain et al. Practical, real-time, full duplex wireless. In *Proc. of MobiCom Conf.*, pages 301–312, September 2011.

[41] V. Jelicic et al. Context-adaptive multimodal wireless sensor network for energy-efficient gas monitoring. *Sensors Journal, IEEE*, 13(1):328–338, January 2013.

[42] Raja Jurdak et al. Opal: A multiradio platform for high throughput wireless sensor networks. *Embedded Systems Letters*, 3(4):121–124, 2011.

[43] Sukun Kim et al. Flush: a reliable bulk transport protocol for multihop wireless networks. In *Proc. of the ACM SenSys Conf.*, pages 351–365, November 2007.

[44] Younghun Kim et al. NAWMS: Nonintrusive autonomous water monitoring system. In *Proc. of the ACM SenSys Conf.*, pages 309–322, November 2008.

[45] Kevin Klues et al. TOSThreads: Thread-safe and non-invasive preemption in TinyOS. In *Proc. of the ACM SenSys Conf.*, pages 127–140, November 2009.

[46] JeongGil Ko et al. Wireless sensor networks for healthcare. *Proc. of the IEEE*, 98(11):1947–1960, November 2010.

[47] JeongGil Ko et al. Beyond interoperability: pushing the performance of sensor network IP stacks. In *Proc. of the ACM SenSys Conf.*, pages 1–11, November 2011.

[48] P. Kumar et al. A hybrid approach to cyber-physical systems verification. In *Proc. of the Design Automation Conf.*, pages 688–696, June 2012.

[49] National Renewable Energy Laboratory. Measurement and instrumentation data center. [Online] `http://www.nrel.gov/midc/hsu/`.

[50] 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.

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

[52] Edward A. Lee. Cyber-physical systems - are computing foundations adequate? October 2006.

[53] Edward A. Lee. Cyber physical systems: Design challenges. In *Proc. of the IEEE ISORC Symp.*, pages 363–369, May 2008.

[54] 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.

[55] Philip Levis. Experiences from a decade of TinyOS development. In *Proc. of the OSDI Symp.*, pages 207–220, October 2012.

[56] Philip Levis and David Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of the ASPLOS Conf.*, pages 85–95, October 2002.

[57] Philip Levis et al. TOSSIM: accurate and scalable simulation of entire TinyOS applications. In *Proc. of the ACM SenSys Conf.*, pages 126–137, November 2003.

[58] Philip Levis et al. TinyOS: An operating system for sensor networks. In *Ambient Intelligence*, pages 115–144, November 2004.

[59] Philip Levis et al. Trickle: A self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *Proc. of the NSDI Symp.*, pages 15–28, March 2004.

[60] 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.

[61] Roman Lim et al. FlockLab: a testbed for distributed, synchronized tracing and profiling of wireless embedded systems. In *Proc. of the IPSN Conf.*, pages 153–166, April 2013.

[62] Guojin Liu et al. Volcanic earthquake timing using wireless sensor networks. In *Proc. of the IPSN Conf.*, pages 91–102, April 2013.

[63] Vincent Liu et al. Ambient backscatter: wireless communication out of thin air. In *Proc. of the ACM SIGCOMM*, pages 39–50, August 2013.

[64] Jiakang Lu et al. The smart thermostat: using occupancy sensors to save energy in homes. In *Proc. of the ACM SenSys Conf.*, pages 211–224, November 2010.

[65] 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.

[66] Geoffrey Mainland, Greg Morrisett, and Matt Welsh. Flask: Staged functional programming for sensor networks. In *Proc. ICFP Conf.*, pages 335–346, September 2008.

[67] Pedro José Marrón et al. TinyCubus: a flexible and adaptive framework sensor networks. In *Proc. of the EWSN Conf.*, pages 278–289, January 2005.

[68] W. P. McCartney and N. Sridhar. Stackless preemptive multi-threading for TinyOS. *Proc. of the DCOSS Conf.*, pages 1–8, June 2011.

[69] 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.

[70] Fredrik Österlind and Adam Dunkels. Approaching the maximum 802.15.4 multi-hop throughput. In *Proc. of the HotEmNets*, June 2008.

[71] Fredrik Österlind et al. Cross-level sensor network simulation with COOJA. In *Proc. of the LCN Conf.*, pages 641–648, November 2006.

[72] Jeongyeup Paek et al. The Tenet architecture for tiered sensor networks. *ACM Transactions on Sensor Networks*, 6(4):1–44, 2010.

[73] Alessandro Pinto. Methods and tools to enable the design and verification of intelligent systems. *AIAA Infotech at Aerospace*, 2012.

[74] 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.

[75] Ragunathan Rajkumar et al. Cyber-physical systems: the next computing revolution. In *Proc. of the Design Automation Conf.*, pages 731–736, June 2010.

[76] Janos Sallai, Akos Ledeczi, and Peter Volgyesi. Acoustic shooter localization with minimal number of single-channel wireless sensor nodes. In *Proceedings of the 9th International Conference on Embedded Networked Sensor Systems*, pages 96–107, November 2011.

[77] Francisco Sant'Anna et al. Safe system-level concurrency on resource-constrained nodes. In *Proc. of the ACM SenSys Conf.*, pages 11:1–11:14, November 2013.

[78] A. Seyedi and B. 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.

[79] Victor Shnayder et al. Simulating the power consumption of large-scale sensor network applications. In *Proc. of the ACM SenSys Conf.*, pages 188–200, November 2004.

[80] Gyula Simon et al. Sensor network-based countersniper system. In *Proc. of the ACM SenSys Conf.*, pages 1–12, November 2014.

[81] 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.

[82] 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.

[83] John A. Stankovic et al. Opportunities and obligations for physical computing systems. *Computer*, 38:23–31, November 2005.

[84] Pablo Suarez, Carl-Gustav Renmarker, Adam Dunkels, and Thiemo Voigt. Increasing ZigBee network lifetime with X-MAC. In *Proc. on the REALWSN Work.*, pages 26–30, April 2008.

[85] 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.

[86] 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.

[87] 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, 2013.

[88] Robert Szewczyk et al. Habitat monitoring with sensor networks. *Commun. ACM*, 47(6):34–40, June 2004.

[89] Janos Sztipanovits et al. Toward a science of cyber-physical system integration. *Proceedings of the IEEE*, 100(1):29–44, 2012.

[90] 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.

[91] S. Tin and A. Lal. Saw-based radioisotope-powered wireless rfid/rf transponder. In *Ultrasonics Symposium*, pages 1498–1501, October 2010.

[92] D. van den Akker and C. Blondia. MultiMAC: A multiple MAC network stack architecture for TinyOS. In *Proc. in the ICCCN Conf.*, pages 1–5, August 2012.

[93] Alex S. Weddell et al. A survey of multi-source energy harvesting systems. In *Proc. of the DATE Conf.*, pages 905–908, March 2013.

[94] 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.

[95] Y. Wu and W. Liu. Routing protocol based on genetic algorithm for energy harvesting-wireless sensor networks. *Wireless Sensor Systems, IET*, 3(2):112–118, July 2013.

[96] Lohit Yerva et al. Grafting energy-harvesting leaves onto the sensornet tree. In *Proc. of the IPSN Conf.*, pages 197–208, April 2012.

[97] 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.

[98] Marco Zimmerling et al. pTunes: Runtime parameter adaptation for low-power MAC protocols. In *Proc. of the IPSN Conf.*, pages 173–184, April 2012.

[99] Zolertia. Z1 platform. [Online] `http://zolertia.com/products/z1`.