UNIVERSITY OF CALIFORNIA,
IRVINE

Scalable Runtime Support for Edge-To-Cloud Integration of Distributed Sensing Systems

DISSERTATION

submitted in partial satisfaction of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

in Electrical and Computer Engineering

by

Tingchou Hung Brett Chien

Dissertation Committee:
Professor Pai H. Chou, Chair
Professor Nader Bagherzadeh
Professor Mohammad Al Faruque

2016

DEDICATION

To my wife, Ke-Yi Chu, and my daughter,
Mina.

TABLE OF CONTENTS

Page

LIST OF FIGURES **vi**

LIST OF TABLES **viii**

LIST OF ALGORITHMS **ix**

ACKNOWLEDGMENTS **x**

CURRICULUM VITAE **xi**

ABSTRACT OF THE DISSERTATION **xiii**

1 Introduction **1**
- 1.1 Wearable Low Power Network-of-Things (NoT) . . . 1
- 1.2 Problem Statement . . . 3
  - 1.2.1 Lightweight and Portability . . . 5
  - 1.2.2 Appropriate Programming Model . . . 5
  - 1.2.3 Low Power with Bluetooth Smart . . . 6
  - 1.2.4 Need for Data Support . . . 7
- 1.3 Contribution . . . 7

2 Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms **8**
- 2.1 Related Work . . . 9
  - 2.1.1 Programming Model . . . 9
  - 2.1.2 Runtime OS support for WSN . . . 12
  - 2.1.3 Virtual Memory . . . 13
  - 2.1.4 File Systems for WSN . . . 14
- 2.2 Overview of Enix . . . 14
  - 2.2.1 Runtime Kernel . . . 15
  - 2.2.2 File System . . . 15
  - 2.2.3 Dynamic Loading Library . . . 16
  - 2.2.4 Utility Tools . . . 16
  - 2.2.5 Code Size . . . 17

2.3 Runtime Components in Enix . . . 17
2.3.1 Cooperative Threads and Scheduler . . . 18
2.3.2 Compiler-Assisted Virtual Memory . . . 21
2.3.3 Dynamic Loading and Run-time Reprogramming . . . 23
2.4 From Enix to Current Proprietary Schedulers . . . 27
2.5 Evaluation and Results . . . 31
2.5.1 Experimental Setup . . . 31
2.5.2 Context Switch Overhead of Different Schedulers . . . 33
2.5.3 Virtual Memory and EcoFS . . . 35
2.5.4 Efficiency of Enix Code Update Scheme . . . 37

3 EcoCast: An Interactive Framework for Parallel Execution of Wireless Sensor Nodes in Multi-Hop Networks 40
3.1 Related Work . . . 40
3.1.1 Remote Firmware Update . . . 40
3.1.2 Shell and Scripting Systems . . . 42
3.1.3 Routing protocols in WSNs . . . 43
3.1.4 Query systems . . . 44
3.2 System Overview . . . 44
3.2.1 Wireless Sensing Platform . . . 44
3.2.2 Execution Flow . . . 47
3.3 Scripting . . . 49
3.3.1 Scripting Language . . . 50
3.3.2 Esh Constructs . . . 54
3.3.3 Method Dispatching and Attribute Access . . . 56
3.4 Compilation and Linking . . . 59
3.4.1 Compilation from C . . . 59
3.4.2 Compilation from Python . . . 60
3.4.3 Incremental Linking . . . 61
3.4.4 Version Control and Patching Scripts . . . 62
3.5 Reprogramming and Execution . . . 63
3.5.1 Execution Mechanism on the Node . . . 63
3.5.2 Group Reprogramming of Nodes . . . 64
3.5.3 Group Execution of Functions on Nodes . . . 66
3.5.4 Background Job Management . . . 67
3.6 Python Bluetooth Low Energy Wrapper . . . 68
3.7 Evaluation . . . 74
3.7.1 Experimental Setup . . . 74
3.7.2 Functional Programming . . . 74
3.7.3 Multi-hop Networking . . . 78
3.7.4 Memory Footprint on the Node . . . 82
3.7.5 Discussion . . . 82

4 A Modular Backend Computing System for Continuous Civil Structural Health Monitoring 83
4.1 Background . . . 83
4.1.1 Event Triggered v.s. Continous Monitoring . . . 83
4.1.2 PipeTECT system for SHM and Water Pipe Monitoring . . . 84
4.2 System . . . 85
4.2.1 DuraMote Smart Sensor . . . 85
4.2.2 Backend System . . . 88
4.3 Evaluation . . . 94
4.3.1 Lessons from Field Experiments . . . 95
4.3.2 Data Traffic Optimization . . . 96
4.3.3 Long-Term Installation at UC Irvine . . . 98

5 BlueRim: Rimware for Enabling Cloud Integration of Networks of Bluetooth Smart Devices 102
5.1 Related Work . . . 102
5.1.1 Cloud Systems . . . 103
5.1.2 HTTP Servers for Sensor Networks . . . 104
5.1.3 Application-Building Tools . . . 104
5.1.4 Device Management . . . 104
5.2 Concepts . . . 105
5.2.1 A Basic BLE Network of Things . . . 105
5.2.2 Rimware with Single Security Domain . . . 106
5.2.3 Rimware with Multiple Security Domains . . . 106
5.3 BlueRim: Cloud and Rimware . . . 108
5.3.1 Cloud Components in Rimware . . . 109
5.3.2 Integration of Rimware with Multiple Security Domains . . . 112
5.4 BlueRim: Physical Subsystem . . . 113
5.4.1 Host-Program Builder . . . 113
5.4.2 NoT Firmware Support . . . 115
5.5 Case Studies . . . 115
5.5.1 Infant ECG . . . 115
5.5.2 Home Automation . . . 116

6 Conclusion 117

Bibliography 120

LIST OF FIGURES

Page

1.1 Enix reference applications. 4
1.2 Relationship between Generic Cloud, Rimware and Network of Things 6

2.1 Number of RF+MCU options for popular ISAs. 10
2.2 The block diagram of Enix. 15
2.3 An example Enix WSN application code: Sense and Transmit. 19
2.4 (a)-(c): Transformation and (d) support for position independent code. 25
2.5 Bridge library and Enix user programs. 26
2.6 Dispatcher on top on a existing schedulor. 27
2.7 Characteristic content format for memory operations. 28
2.8 Characteristic content format for function operations. 29
2.9 Access SFR and memory in the MCU. 29
2.10 Ram code: Upload and execute. 30
2.11 Calling existing function calls. 30
2.12 (a) EcoSpire node: 23×50 mm^2; (b) EcoSpire simple node, 13×20 mm^2. 31
2.13 Power measurement modules. 32
2.14 SD card reading using different block lengths. 37

3.1 System overview. 45
3.2 Execution Flow of EcoCast. 47
3.3 Reprogramming scenario 65
3.4 Function execution sketch 66
3.5 Profile handler skeleton and a default profile handler implementation. 69
3.6 PyBLE usage 70
3.7 A simple perpheral is described by PyBLE. 71
3.8 An example implementation of a exisintg acceleration profile. 72
3.9 A BLE battery service hanlder implementation. 73
3.10 Reprogramming Latency 76
3.11 Response time of ADC parallel execution of EcoCast 76
3.12 Response time of serial execution and parallel execution of EcoCast 77
3.13 Ratio of different phases to execution time of ADC via parallel execution of EcoCast. 78
3.14 Union of connectivity graphs 79
3.15 (a) Latency of finding a path for each node, (b) Response time of calling LIGHTING function. 80
3.16 The latency of adding functions 81

3.17 The latency of calling functions 81

4.1 Gopher 86
4.2 Accelerometer 86
4.3 Roocas 87
4.4 Overview of Our Backend System 89
4.5 Campus 92
4.6 Using Smartphone to show the monitoring system 92
4.7 Google Map Service Integration 94
4.8 Locations of deployed sensors 97
4.9 Engineering Tower 98
4.10 Engineering Hall 99
4.11 CALIT2 99
4.12 Engineering Gateway 99
4.13 AIRB 100

5.1 Relationship between Generic Cloud, Rimware and Network of Things 105
5.2 Customer, BlueRim and Vendors 107
5.3 Components in Rimware 108
5.4 Workflow of Host Program Builder 114

LIST OF TABLES

Page

1.1 Comparison between common sensor platforms 3

2.1 Market share of Microcontroller ISAs 9
2.2 Comparison between WSNs operating systems. 10
2.3 Context switch overhead comparison between algorithms for selecting the next running thread. Unit: μs. 33
2.4 Context switch overhead comparison between different scheduler implementation. Unit: μs. 33
2.5 Context switch overhead comparison between Enix and μC/OS-II by averaging 60,000 context switches. 34
2.6 Code and data size comparison between different scheduler implementation. Unit: bytes 34
2.7 Code and data size comparison between Enix, FreeRTOS and μC/OS-II. Unit: bytes 35
2.8 I/O speed comparison between flash chips. Units: Kbytes/s. 36
2.9 The energy consumption comparison between different flash chips. unit: mJ. . . . 36
2.10 SD card comparison between manufacturers. 36
2.11 Runtime reprogramming code size with/without Enix. Unit: bytes. 39
2.12 Update code size using VCDIFF delta compression. Units: bytes 39
2.13 The energy consumption comparison between SD card and RF (Full Speed). Units: mJ. 39

3.1 Comparison of partial image replacement techniques. 41
3.2 Comparison of scripting systems for WSN. 42
3.3 Scoping of symbols 54
3.4 Summary of EcoCast commands 57
3.5 Test cases of functional programming 74
3.6 The Latency of topology discovery phase 80
3.7 Comparison of Memory footprint (bytes) 82

4.1 Improvements in Data Dissemination Subsystem 97

LIST OF ALGORITHMS

		Page
1	Fast algorithm to get the next running thread (Priority-Based).	21
2	Fast algorithm to get next running thread (Round-Robin).	21

ACKNOWLEDGMENTS

First of all, I would like to express my deepest gratitude to my advisor, Dr. Pai H. Chou. My PhD study would not be completed without his guidance and encouragment.

Secondly, I am grateful to have my committee members, professor Nader Bagherzadeh and professor Mohammad Al Faruque. I am honored to have them in my committee. I also wish to send my best regards to all my colleagues in Professor Chou's group. It is a great pleasure to meet and work with young and talented people from all around the world. I would like to say thank you to all of you, Sehwan Kim, Eunbae Yoon, Chengjia Huo, Jun Luan, and Seung-Jae Lee.

The Works are also contributed from following people. In Enix project, Yu-Ting Chien has done the first version of the implementation In EcoCast Yi-Hsuan Tu, Yen-Chiu Li are having the first working prototype that I can improve base on their solid fundation.

I am truely blessed to have my family around me. Thanks for my parents to support me through the study. Thanks for my family members in United States to let me never feel alone. Thanks for my wife and daughter, they are my source of joy and supporting me through all these years.

This work was sponsored in part by a National Institute of Standards and Technology (NIST) Technology Innovation Program (TIP) Grant 080058, National Science Foundation (NSF) grant CBET-0933694, a National Institute of Health (NIH) STTR contract 1R41HL112435-00A1 through a subcontract from Pulsentry, Inc. ("ECG Device for LQTS Screening in Newborns, Phase I"), an NIH SBIR contract 1R43HD074379-01 through a subcontract from Intelligent Optical Systems, Inc. (IOS) ("Infant Sleep Environment Monitoring System (SEMS) for SIDS (Phase 1)"), an NIH STTR contract 1R41HD079051-01A1 from IOS ("Pulse Oximeter for Newborn Screening"), and an NIH STTR contract HL112435 through subcontract QT001 from QT Medical, Inc. ("ECG Device for LQTS Screening in Newborns, Phase II").

CURRICULUM VITAE

Tingchou Hung Brett Chien

EDUCATION

Doctor of Philosophy in Engineering and Computer Science **2016**
University of California, Irvine *Irvine, California*

Bachelor of Science in Computer Science and Information Engineering **2004**
National Chiao-Tung University *Hsinchu, Taiwan*

RESEARCH EXPERIENCE

Graduate Research Assistant **2007–2012**
University of California, Irvine *Irvine, California*

TEACHING EXPERIENCE

Teaching Assistant **2009–2010**
University name *City, State*

REFEREED JOURNAL PUBLICATIONS

1. Chengjia Huo, Ting-Chou Chien, Pai H. Chou, "Middleware for IoT-Cloud Integration across Application Domains,", in *IEEE Design & Test, Volume 31, Issue 3, June 2014. pp. 21–31. doi: 10.1109/MDAT.2014.2314602.*

REFEREED CONFERENCE PUBLICATIONS

1. Jun Luan, Ting-Chou Chien, Seungjae Lee, Pai H. Chou, "HANDIO: A Wireless Hand Gesture Recognizer Based on Muscle-Tension and Inertial Sensing," in *Proceedings of the Global Communications Conference: Selected Areas in Communications: E-Health (âĂŹGCâĂŹ 15 - SAC - E-HealthâĂŹ)*, San Diego, CA, USA, December 6-10, 2015.

2. Ting-Chou Chien, Chengjia Huo, Pai H. Chou, "A Modular Backend Computing System for Continuous Civil Structural Health Monitoring," in *Proceedings of the Nondestructive Characterization for Composite Materials, Aerospace Engineering, Civil Infrastructure, and Homeland Security 2014 , Volume 9063, , March 9-13, 2014. doi: 10.1117/12.2045264*

3. Marco Torbol, Sehwan Kim, Ting-Chou Chien, Masanobu Shinozuka, "Hybrid networking sensing system for structural health monitoring of a concrete cable-stayed bridge", in *Proc. SPIE 8692, Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems 2013, 86920C (April 19, 2013); doi:10.1117/12.2009705*

4. Sehwan Kim, Eunbae Yoon, Ting-Chou Chien, Hadil Mustafa, Pai H. Chou, Masanobu Shinozuka, "Smart Wireless Sensor System for Lifeline Health Monitoring under a Disaster Event", in *Proc. SPIE 7983, Nondestructive Characterization for Composite Materials, Aerospace Engineering, Civil Infrastructure, and Homeland Security 2011*

5. Yi-Hsuan Tu, Yen-Chiu Li, Ting-Chou Chien, Pai H. Chou, "EcoCast: Interactive, Object-Oriented Macroprogramming for Networks of Ultra-Compact Wireless Sensor Nodes," in *Proc. the 10th International Conference on Information Processing in Sensor Networks (IPSN 2011)*, Chicago, IL, USA, April 12-14, 2011. pp. 366–377.

6. Yu-Ting Chen, Ting-Chou Chien, Pai H. Chou, "Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms," in *Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys 2010)*, Zurich, Switzerland, November 3-5, 2010. pp. 183-196.

PATENTS

1. Meng-Shiuan Pan, Yen-Ann Chen, Ting-Chou Chien, Yueh-Feng Lee, and Yu-Chee Tseng, "Automatic Lighting Control System And Method", *US 11/950,429*

ABSTRACT OF THE DISSERTATION

Scalable Runtime Support for Edge-To-Cloud Integration of Distributed Sensing Systems

By

Tingchou Hung Brett Chien

Doctor of Philosophy in Electrical and Computer Engineering

University of California, Irvine, 2016

Professor Pai H. Chou, Chair

While Internet-Of-Things (IoT) has drawn more attention to researchers and the public, to build a complete system from the edge sensing units to the cloud services requires massive amount of efforts. Researchers with strong interests in collected information are often lost in various technologies, including distributed sensing embedded systems, bridge devices between Internet and local network, and data backend services.

This work takes a cross-system, script-based, and semantic-enhanced approach to address the problem of lacking suitable runtime supports. We proposed a threaded code runtime support for edge sensing systems, a script based wrapper on Physical-to-Cyber bridges, and scalable middleware into the backend services.

With proposed runtime supports, we are able to apply distributed sensing systems into real world applications quickly and explorer insights from collected information. As a result, a building structure monitoring system is installed and allow civil researchers to develop algorithms to prevent disaster events. Body area sensing systems such as ECG monitoring, CO_2 detection, and body movement are developed. This enables baby screening and detect potential heart problems. The results have shown that with proposed runtime supports applications can be realized quickly and scalable.

Chapter 1

Introduction

The Internet of Things (IoT) has been receiving growing attention in recent years as the next wave of computing revolution made possible by low-cost, miniature low-power systems-on-chip (SoC) with computing and communication capabilities. Among many applications, we focus on wearable medical devices, home automation, and personal tags as representative examples. Three recurring themes in building IoT applications are low power, management, and data support.

1.1 Wearable Low Power Network-of-Things (NoT)

The term *network-of-things* (NoT) refers to a locally connected network of embedded systems with (sensing, actuation) interfaces to the physical world, whereas the *Internet of Things* (IoT) refers to the interconnected NoTs bridged by the Internet Protocol (IP). The NoT can also be refferenced as *Wireless Sensor Networks* (WSN) in the early stage of NoT.

Figure 1.1 shows three scenarios that WSN researchers often encounter. The first issue is the size and cost problem. Current WSN platforms, shown in Table 1.1, for example, Mica2 [29] and MicaZ [1], have a development board with two AA batteries that may be "small" when compared

to a computer but are actually, big, heavy, and expensive to wear and embed for many real-life wireless sensing applications. Wearable sensing system, including both human and animal worn ones, impose possibly the most stringent constraints on the size and weight of the entire system, as shown in Figure 1.1(a)

To enable better fitness to wearable applications, some ultra-compact wireless sensor platforms have been developed [68, 87]. The Eco node, which uses an 8051/8052-compatible MCU core, is only $1cm^3$ in volume and weighs under 2 grams, making it most competitive in terms of cost and lightweight metrics. However, it is also highly constrained in terms of the amount of memory and computing ability, which pose new challenges on the operating system design.

Besides the ultra compact wireless sensor platforms, a medical platfrom has been developed,

To provide effective management of tight constraints on hardware resources and to enable runtime support of wireless sensor nodes, a lightweight system kernel that is designed specifically for WSN has become increasingly important. Most of people think that an operating system may cause additional overhead and power consumption especially for resource-constrained sensing systems. However, an operating system may actually improve efficiency by supporting power management, memory management, task management, and multi-threading.

Figure 1.1(c) shows typical capabilities of runtime operating systems. In addition to the basic scheduling for event-driven or multi-threading models, several other types of runtime support are also becoming important for wireless sensing applications. For example, virtual memory enables the MCU to execute larger, more sophisticated programs than its physical memory alone allows. Another important type of support is remote programming, as it may be nearly impossible to update firmware of sensor nodes through wired interfaces after they have been deployed in large numbers. For this reason, a loader that supports dynamic reprogramming becomes an important issue when designing an operating system for WSNs. Users expect that firmware update can be done using a host PC shell environment. Code image will be sent through RF to remote sensor nodes and loaded

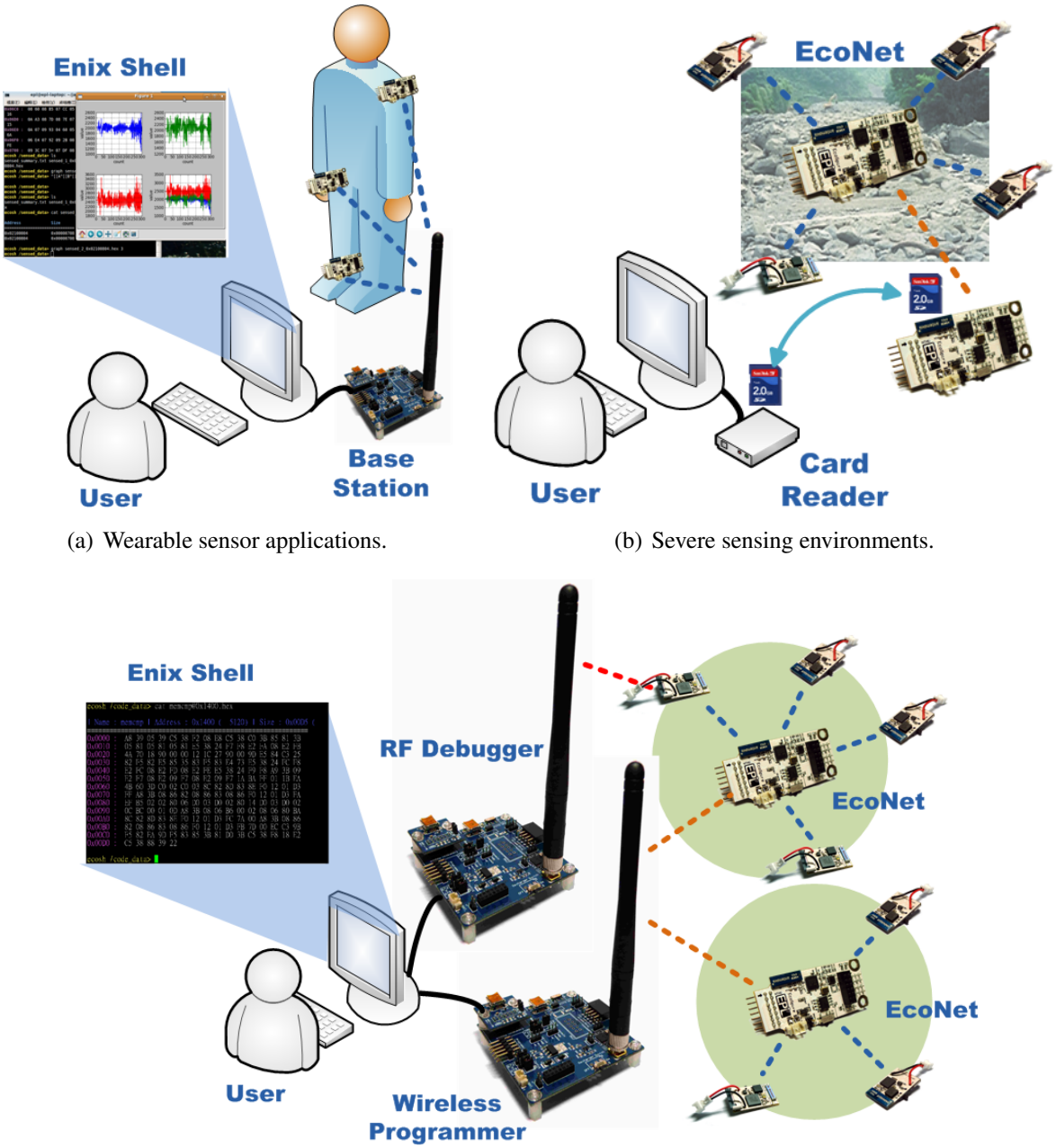
Table 1.1: Comparison between common sensor platforms

	EcoSpire simple node	EcoSpire super node	Mica2	MicaZ	Telos Rev.B
Size (incl. batt.)	2×1.3×0.8 cm	5×2.3×0.8 cm	3.2×5.7×1.5 cm	3.2×5.7×1.5 cm	3.2×8×2 cm
Microcontroller					
Type	NRF24LE1	NRF24LE1	ATmega128	ATmega128	MSP430
Frequency(MHz)	16	16	16	16	8
Data mem.(KB)	1	1	4	4	2
Code mem.(KB)	16	16	128	128	60
External Storage					
Ext. mem.	no	SD card(2GB)	Flash (512KB)	Flash (512KB)	Flash (512KB)
Communication					
Radio	NRF24L01+ (Integrated)	NRF24L01+ (Integrated)	CC1000	CC2420	CC2420
Data rate	2Mbps	2Mbps	38.4Kbps	250Kbps	250Kbps
Protocol	Enhanced Shock Burst	Enhanced Shock Burst	SmartRF	Zigbee	Zigbee
Antenna	Build-in	Build-in or External	External	External	External
Power source					
Type	Li-ion	Li-ion	2 AA	2 AA	2 AA
Capacity (mAh)	80	80	4000	4000	4000
Rechargeable	yes	yes	no	no	no
Others					
Sensors	On-board Sensors or Separate module	On-board Sensors or Separate module	Separate module	Separate module	On-board Sensors
Host & prog. i/f	USB or UART	USB or UART	UART	UART	USB

by a dynamic program loader at runtime. Another issue that arises from remote reprogramming is the size of the code image. In order to reduce the power consumption, sensor nodes should reduce the RF transmission as more as possible; moreover, they should also avoid idle-listening, since that consumes twice the power as the highest RF transmission. A good run-time system may also provide an efficient mechanism to reduce the image size in remote reprogramming. Some common schemes such as patch generation and compression can achieve the objective, but these also increase the runtime overhead and consume considerable runtime memory. As a result, a better solution to the problem must be presented.

1.2 Problem Statement

Our goal is to implement a new operating system for wireless sensor platforms. Specifically, we target wireless sensor systems that are ultra-compact with a limited number of general-purpose I/O (GPIO) pins and a small amount of code and data memory. Such platforms typically include



(a) Wearable sensor applications.

(b) Severe sensing environments.

(c) Remote reprogramming and debugging.

Figure 1.1: Enix reference applications.

on-board sensors to collect data and exchange data through RF; common interfaces such as UART, I2C, and SPI are also available. The external non-volatile memory such as serial flash and Micro-SD card can be connected to the sensor device through SPI. In the following subsections, we list the requirements of a WSN OS design.

1.2.1 Lightweight and Portability

The operating system should be lightweight and portable enough to run on some of the most resource-constrained wireless sensor platforms with different MCU ISAs. Low memory and power consumption must be achieved by utilizing only limited resources in order to increasing the life time of wireless sensor devices. A portable interface and the reduction of assembly code implementation enable the operating system to be ported to other MCU ISAs.

1.2.2 Appropriate Programming Model

An appropriate programming model not only facilitates software development but also promotes good programming practices. Event-driven programming model is widely used in WSN but can be unstructured and difficult to use [25, 6, 85, 2]. Consequently, an easy-to-use threading structure that is applicable to event-based WSN applications with little runtime overhead is desired. Context switching and the scheduling policy are the two main sources of run-time overhead of multi-threaded programming; the efficiency of the scheduler must be improved by fast algorithms instead of linear searching for the next running thread; the overhead of context switch can also be reduced by simply storing the critical registers onto the stack and avoiding using the slow external memory.

Several recent works are suggesting *scripting* as a new trend towards programming and control of wireless sensor networks (WSN). Scripting languages started out as textual commands in *shells*,

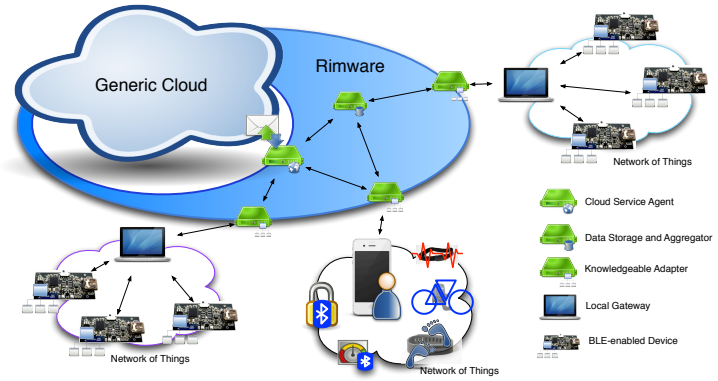


Figure 1.2: Relationship between Generic Cloud, Rimware and Network of Things

or command-line interpreters (CLI), for interactive invocation of utility programs for everyday tasks from file system manipulation and text editing to application-level programming and system administration. Batch shell commands with the addition of control flow constructs evolved into scripting languages, such as awk, perl, sh/csh, tcl, ruby, and many others. Scripting languages have been credited for helping put together many pieces of the world-wide web, including news-group servers, CGI scripts, PHP, and many graphical user interfaces. Ousterhout also demonstrated that scripting can enable programmers to achieve 10× the productivity over *system programming* languages such as C or Java [65].

1.2.3 Low Power with Bluetooth Smart

Low power consumption is a mandatory feature in IoT. Bluetooth Smart, also known as Bluetooth 4.0 Low Energy Technology (BLE), stands out as the lowest in average power consumption among many wireless standards. A BLE node can last for one year on a CR2032 coin-cell battery. BLE also enables users to interact with these IoT nodes using their smartmobiles directly without infrastructure support or dongles, which can be inconvenient. For these reasons, we believe BLE deserves high-priority support in IoT.

1.2.4 Need for Data Support

IoT applications that may need support and automation for data handling include wireless sensor networks (WSN) and medical devices. They act as data sources whose data may need to be uploaded for post-processing, analysis, archive, and disseminated in a controlled way. However, many of today's IoT systems handle data in an ad hoc manner with many hardwired operations, making them difficult to extend or generalize. A general framework is sorely needed for supporting these data operations in a systematic, secure, and access-controlled way. Data should be made available as a service to support composition with other services.

1.3 Contribution

In this work, an edge-to-cloud runtime supports have been proposed from embedded system level to Web services integration. In the embedded systems, a programming model based on threaded code has been proposed and implemented. This model can be incorperated into other existing scheduler used by SoCs. For the connectity programming in gateway devices, a python and swift language to operate Bluetooth Smart has been implemented to wrap around the wireless transmission to allow users to focus on data. Than a backend integrated system has been implemented to demostrate a possible medical sensing system can be intergrated to discover new possiblities for remote healthcare infrastructure.

Chapter 2

Enix: A Lightweight Dynamic Operating System for Tightly Constrained Wireless Sensor Platforms

Enix is a lightweight dynamic operating system for tightly constrained platforms for wireless sensor networks (WSN). Enix provides a cooperative threading model, which is applicable to event-based WSN applications with little run-time overhead. Virtual memory is supported with the assistance of the compiler, so that the sensor platforms can execute code larger than the physical code memory they have. To enable firmware update for deployed sensor nodes, Enix supports remote reprogramming. The commonly used library and the main logical structure are separated; each sensor device has a copy of the dynamic loading library in the Micro-SD card, and therefore only the main function and user-defined subroutines should be updated through RF. A lightweight, efficient file system named EcoFS is also included in Enix. The code size and data size of Enix with full-function including EcoFS are 8 KB and 512 bytes, respectively, enabling Enix to run on many RF-enabled systems-on-chip that cannot run most other WSN OSs.

Table 2.1: Market share of Microcontroller ISAs

Co.	Core	Sh.
Intel	8051	19%
Renesas	740, H8/S, M32R	17%
Freescale	68XX	15%
PIC	PIC	12%
ARM	ARM	10%

Co.	Core	Sh.
NEC	V850, 78K0, K3/K4	9%
ST	Proprietary 8-b	6%
Atmel	AVR	3%
Infineon	C16X	3%
Others	Others	6%

2.1 Related Work

Wireless sensor networks are composed of sensor nodes and base stations. A sensor node is typically small in physical size, low cost, small in code and data memory, and limited in computing capability and battery energy. Table 2.1 lists the instruction set architectures (ISA) of the most popular MCUs. Some real-time operating systems (RTOS) that are able to run on such a resource-constrained platform include μC/OS-II [52] and FreeRTOS [75], both of which support preemptive multi-threading with round-robin or priority-based schedulers. These RTOSs are indeed lightweight and well designed, but they are not suited for wireless sensor networks due to the lack of support for features such as runtime code update, power management, and resource control capabilities. In response, researchers have proposed WSN OSs that provide support specifically for WSN applications. Table 2.2 shows a comparison between existing OSs aimed at WSNs. However, it seems ironic that most WSN OSs to date are for either Atmel (3%) and MSP430 (a fraction of 6% "Others"), leaving the most popular 8051 (19%) and most other ISAs unsupported. When considering integrated RF-MCUs, the number of options for the 8051 dominates all other ISAs combined by 4:1, as shown in Fig. 2.1.

2.1.1 Programming Model

TinyOS [54] is a widely used runtime system for WSNs. It uses a special language called nesC [22] to describe the software components that form a sensor system with event-driven semantics. The application code and the runtime library are then compiled into one monolithic executable. Several

Table 2.2: Comparison between WSNs operating systems.

OS	Enix	TinyOS	SOS	Contiki	MANTIS OS	t-kernel	RETOS	LiteOS	Nano-RK
Platform	Nordic nRF24LE1	ATmega128L &MSP430	ATmega128L	ATmega128L & MSP430	ATmega128L	ATmega128L	ATmega128L & MSP430	ATmega128L	ATmega128L
Programming Model	Thread	Event	Event	Event & Thread	Thread	Thread	Thread	Thread	Thread
Real-time Support	△ [1]	△		◯	◯	◯	◯	◯	◯
Dual Mode Operation[9]	◯		◯	◯	◯	◯	◯	◯	
Remote Update	◯	△	◯	◯	◯		◯	◯	
Dynamic Loading	◯ [2]	△	◯ [2]	◯ [3]			◯ [3]	◯ [3]	
Protection[10]		△				◯			
Virtual Memory	◯ [4]	△				◯ [5]			
File System	◯	△						◯	
Network Abstraction[11]	◯	△					◯		◯
Code Size (Bytes) [6]	8,138	20,924 [8]	20,464	3,874	14,000	28,864	20,394	30,822	10,000
Data Size (Bytes) [7]	512	597	1536	512	512	512	945	1,633	2,000

[1] △ means optional components.
[2] achieved using PIC.
[3] achieved using runtime relocation.
[4] supports code virtual memory only.
[5] supports both code and data paging.
[6] The code size including the basic kernel and the ◯ components, excluded △s.
[7] lists the smallest data memory required to startup OS, at least one thread in multi-threaded model.
[8] TinyOS code size is compiling from v1.1.15, with various sensor drivers and a network module, but excluding storage system and remote programming.
[9] means the kernel code is not writable by user code
[10] threads are protected from each other's access
[11] OS provides layers of API

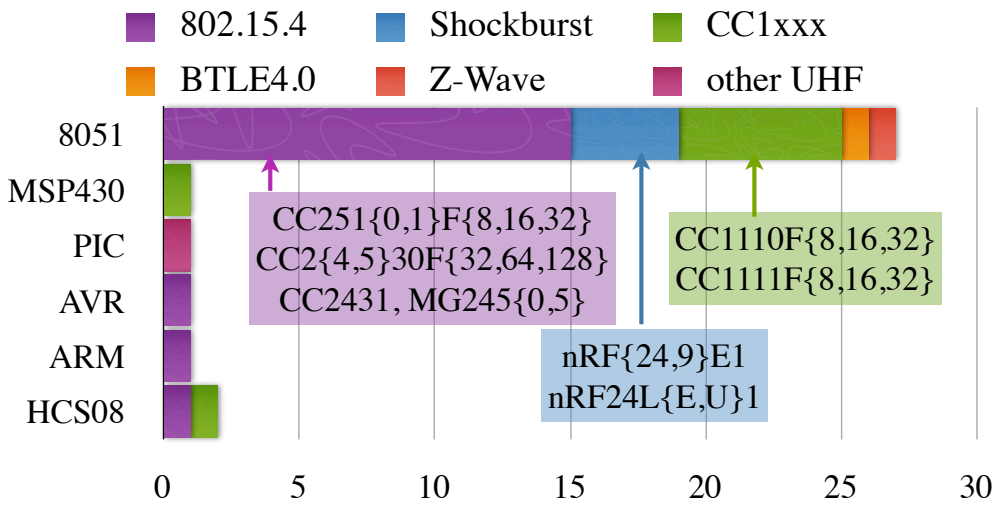


Figure 2.1: Number of RF+MCU options for popular ISAs.

event-driven runtime-support systems have been developed for wireless sensor networks and applications with similar characteristics, including SOS [27] and Contiki [20]. The processes of these OSs are implemented as event handlers that *run to completion* without preemption. Therefore, event handlers in event-driven models may share the same stack to reserve insufficient memory space. Event-driven programming is based on cooperative multitasking, which may be good for tiny, single-processor embedded devices, but users have to perform stack management manually, and as a result, the code can become difficult to read and maintain. Some WSN OSs provide preemptive multi-threading [6, 24, 20, 11, 10, 46]. Multi-threaded programming models can be easier

to learn compared to event-driven ones, and the code can be more readable and maintainable. In severely memory- and power-constrained environments, however, a multi-threaded model has several disadvantages. For example, it occupies a large part of the memory resources, spends more CPU time and consumes more battery energy due to context-switching overhead.

Preemptive multi-threading extensions [17] to TinyOS have been proposed to avoid long-running tasks. Various concurrency methods has been considered, including preemptive tasks, fibers and virtual machine threads. Later, TinyOS v2.x incorporated the multi-threading concept in TOSThreads [46]. They consist of kernel threads and application threads. Only application threads can be preempted, while kernel threads are non-preemptible since they are for blocking system calls. Kernel threads are activated only through message passing.

Protothreads [21] allow users to write event-driven programs in a threading style, with the memory overhead of two bytes per Protothread. The concept is similar to C co-routines [47] in that they implement "return and continue" by using C-switch expansion. Protothreads are limited in that auto variables or local states cannot be stored across a blocking wait, and the C-switch implementation may lead to increased code size in the form of a table lookup and a jump, which also incur overhead at runtime. The runtime complexity is proportional to the number of yield points in a Protothread.

Our Enix OS emphasizes cooperative threading, which guarantees that no thread will have to yield control unexpectedly [25], and Enix threads work similarly to co-routines but they are implemented in a mix of C and assembly for smaller code size and better execution efficiency. Swapping between threads in Enix is a real context-switch operation, not just a C-switch. It provides automatic stack management and incurs little runtime overhead compared to preemptive multi-threading. In addition to cooperative threading, Enix also supports lightweight preemptive multi-threading for real-time scheduling. For the power consumed by context switches, it has been shown for MANTIS OS that multi-threading and energy efficiency need not be mutually exclusive when an effective sleeping mechanism is used to reduce context-switching overhead [6, 17].

2.1.2 Runtime OS support for WSN

In real-world wireless sensor networks, the deployed sensor nodes must have the abilities to manage the tasks and resources at run-time. Run-time reconfiguration and reprogramming also become important issues in WSN OS design. TinyOS, as mentioned before, produces a single image in which the kernel and applications are statically linked. Thus, updating code means whole-system image replacement. To support efficient runtime remote reprogramming for TinyOS, a virtual machine named Maté [53] has been proposed. Using Maté or other virtual machines for WSNs [49, 60], code can be distributed and configured at run time. The drawbacks of running a virtual machine on a sensor node include runtime overhead of the virtual machine interpreter and higher energy consumption.

SOS [27] is another event-driven OS but consists of dynamically loaded modules and a common kernel. The modules are position independent code (PIC) binaries that implement specific tasks or functions. This modularized design is quite flexible, but the interaction between modules may incur high runtime overhead, and the module must be implemented in a fixed format, which increases the overall code size on SOS.

Contiki [20], RETOS [11] and LiteOS [10] provide dynamic loading by runtime relocation, rather than relying on position independence. The application binary to be combined with relocation information must be relocated or linked at runtime [18] before loading into the program memory. Thus, a sizable data buffer is required in order to relocate in space.

Our Enix OS supports dynamic loading using kernel-supported PIC, which is easy to port to other platforms. The dynamic library is pre-linked to the kernel with minor modification. Hence, our runtime overhead and additional buffer are reduced compared to the runtime relocation approach.

2.1.3 Virtual Memory

To fully utilize the memory of a tightly constrained wireless sensor platform, some researchers propose software virtual memory on MMU-less embedded systems. One approach is code modification by either using a compiler or constructing an additional converter. SNACK-pop [67] provides a framework for compiler-assisted demand code paging with static call graph analysis and optimization. The input Executable and Linking Format (ELF) [37] file is analyzed and translated into an executable image such that every call/return is modified to call the page manager. Choudhuri and Givargis [13] showed that data segments have a greater need to be paged than code segments, and therefore they propose a data paging scheme with an adjustable page size based on an application-level virtual memory library and a virtual memory-aware assembler. The t-kernel [24] is the first WSN OS that provides virtual memory for both code and data segments with additional memory protection. Besides software virtual memory approach, MEMMU [5] proposes a new software-based on-line memory expansion technique [4] that requires no secondary storage. Therefore, it improves performance and minimizes power consumption comparing to the above approach. However, MEMMU introduces about 4 KB of code size overhead and requires at least 512 bytes of data memory. Besides dynamic loading, Enix also supplies software segmented virtual memory by code modification, and uses a Micro-SD card as secondary storage for the convenience of installing virtual code segments using a host PC without requiring a specialized programmer. Enix does not provide virtual memory for data segments because of the high runtime overhead, but it does support a data memory allocation scheme to fully utilize the limited data memory. Furthermore, VM support in Enix consumes only 886 bytes of code memory and 256 bytes of data memory.

2.1.4 File Systems for WSN

LiteFS, a subsystem of LiteOS [10], provides a hierarchical, Unix-like file system that supports both directories and files. General-purpose file systems such as FAT, Ext2 may be interoperable but may be unsuitable for WSNs due to the relatively large code size and a great deal of data memory space used to store the hierarchical data structures. A more severe problem is the poor performance: in our own experience, a popular FAT file system was able to log at most 35 samples per second on a Telos-class node. Another issue is that the general file format becomes insufficient to store WSN data due to the absence of fast query support.

Some file systems or databases have been customized for WSNs. ELF [16] uses NOR flash to implement a log-structured file system. MicroHash [90], TinyDB [57] and FlashDB [62] use a lightweight index structure or database that works on wireless sensor nodes. Due to supporting high-performance indexing and searching capabilities, overhead of in-memory data structures is inevitable on these systems.

The Coffee file system adopted in ContikiOS is a log-structured, flash-based file system [84]. Capsule [59] covers the abilities mentioned above. It provides the abstraction of typed storage objects to applications, including streams, indexes, stacks, and queues. The composition of different objects may satisfy various requirements of WSNs. However, the high capacity parallel NAND flash used by Capsule makes use of many general-purpose input/output (GPIO) ports, and therefore it does not work on small-sized MCUs with few available I/O ports.

2.2 Overview of Enix

Fig. 2.2 shows the block diagram of Enix, which currently runs on a wireless sensor platform called EcoSpire [12]. The Enix OS contains four major components: runtime kernel, file system,

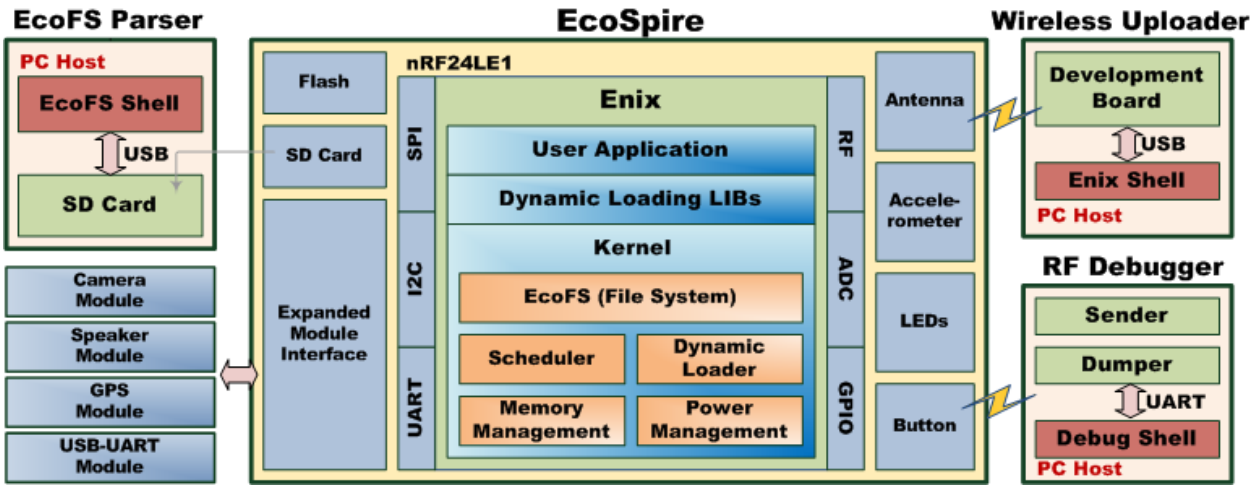


Figure 2.2: The block diagram of Enix.

dynamic loading library, and utility tools. Although the file system is one of the modules in the runtime kernel, it provides more support for WSN applications. Consequently, we separate the file system as an individual component of Enix.

2.2.1 Runtime Kernel

The first component of Enix is the *runtime kernel*, which manages hardware resources and supports run-time reconfiguration. With a *cooperative thread scheduler*, developers can write multi-threaded programs instead of being limited to single-threaded programming. A *wireless code image update manager* and a *dynamic loader* are also included to enable remote reprogramming, so that the deployed sensor nodes can be easily updated over RF.

2.2.2 File System

EcoFS, the file system of Enix, is a configurable and lightweight storage system using a Micro-SD card or an MMC card as the storage medium. Generic file abstraction may not be well suited for WSNs. We realized that only few types of data need to be saved, and we divide EcoFS according

to the different usage patterns: *code data*, *preferences*, *network data*, and *sensed data*. The code data block stores the binary code segments that can be loaded efficiently by the runtime loader into the code memory on demand. The preferences are the key-value pairs for storing the node's status and settings. The requirement of preferences is fast searching with modifications enabled. Because of the limited data memory and the wish to provide a simple network abstraction, the routing table may be maintained in EcoFS. Network data such as routing tables or the roles of the adjacent nodes must be enumerable and may be changeable if the network topology is dynamic. Last but not least, the sensing data block is used for logging collected data in harsh environments when real-time transmission is not viable. This append-only storage type must be fast, power-efficient, and reliable to meet the requirements of high-sampling-rate WSN applications.

2.2.3 Dynamic Loading Library

The third component of Enix is the dynamic loading library called ELIB. ELIB is a special library that is preprocessed by the host PC tools. Most of the commonly used library functions are collected and transformed into *segments* that comprise ELIB. Each segment corresponds to the position-independent code of a function with a unique virtual address, namely where the segment is located in the secondary storage. Due to the constrained memory resource of compact wireless sensor devices and the high energy consumption of the RF transceiver, making use of ELIB enables software virtual memory and reduces the transmission size of runtime reprogramming. A full ELIB library now takes 28K (12KB without standard C library).

2.2.4 Utility Tools

The last part of Enix is the host PC's *utility tools*, including a wireless reprogramming and debugging shell, a file system parser shell, complier and linker. With the host PC's assistance, heavier computation and compiling tasks can be done on the host PC to reduce the runtime overhead on

tightly constrained nodes. Rather than processing delayed linking at run-time as ELF does, the ELIB building tool constructs a position-independent library at link-time. Therefore, the run-time loading burden on the MCU of these compact sensor nodes is reduced significantly, and no additional code is required. Another tool is the file system shell, called the *EcoFS Shell*, which provides an interactive environment for users to manipulate specially formatted data of EcoFS simply and conveniently. This assistance saves the user from wasting time on troublesome tasks such as reading, modifying, and writing secondary storage devices directly.

2.2.5 Code Size

The non-swappable part of the system takes around 8KB of compiled code, including the thread scheduler, low-level drivers for SD and RF, file system driver, remote programming, the main function, and some basic library needed by above objects. Other utility library code can be saved in ELIB on the SD card and are loaded on-demand. A detail breakdown for Enix compiled code is as follows:

	Enix Scheduler	1023
	Enix Storage and RF driver	1696
	EcoFS	2316
	Remote reprogramming	660
	Essential C library	1130
+)	User logical structure	203
	Total	7028 bytes

2.3 Runtime Components in Enix

This section describes the kernel components in Enix. They are the cooperative threads with the run-time scheduler, compiler-assisted virtual memory, and dynamic loading and runtime reprogramming support.

2.3.1 Cooperative Threads and Scheduler

The *cooperative threading* model has the characteristic that a context switch occurs only when the current running thread calls a yield or sleep function. Thus, the overhead of context switching and stack usage are lower than that of preemptive multi-threading and is more appropriate for tightly constrained wireless sensor platforms. Here, we describe an efficient way to implement cooperative threads with low context-switching overhead and that consumes little code and data memory. In addition, two popular scheduling policies, namely priority-based and round-robin, are also presented to provide the adaptive abilities of Enix for supporting different WSN applications.

Multi-points Setjmp/Longjmp

The system calls setjmp and longjmp are commonly used for jumping between subroutines. To achieve the inter-subroutines jump, the setjmp function stores the program counter and stack pointer into a jump buffer. When longjmp function is called from another subroutine, the data in the jump buffer is restored, and then the program returns to the previous setjmp point. It is worth mentioning that the used registers are pushed on the stack before the setjmp function is called, and therefore the local variables can also be restored after setjmp. However, this approach does not support jumping forward or backward between multiple functions as required by coroutines. First, it provides a single-direction jump only from the longjmp point to setjmp point according to the jump buffer; second, stack overwriting happens while the previous setjmp point calls the cascaded function that may modify the stack data of later setjmp points.

To achieve cooperative threading, the ideas of setjmp and longjmp are taken. Each never-returned subroutine represents a cooperative thread, which has its own stack and context buffer for storing critical data. A cooperative thread may yield at any specific point to invoke the scheduler and resume another cooperative thread. Each yielding call automatically pushes the local variables on the stack and records the program counter and stack pointer in the context buffer. The

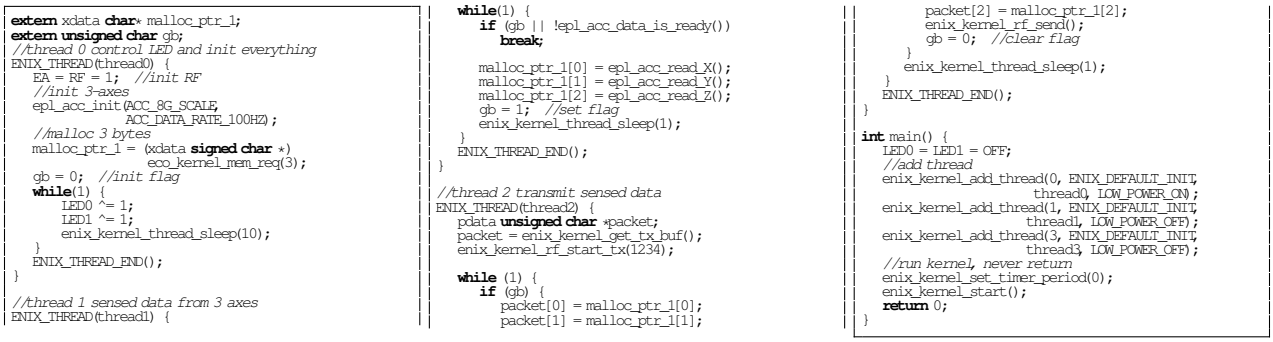


Figure 2.3: An example Enix WSN application code: Sense and Transmit.

resuming process simply restores the saved data from the context buffer and the stack. As long as the context-switching point is determinable, only necessary data will be stored on the stack. Therefore, the per-thread stack does not require much memory and is adaptive to the number of threads. The maximum number of cooperative threads is seven in the current implementation of Enix. This approach to cooperative threads allows subroutines to suspend and resume execution at specific locations without being concerned with stack and thread states. Moreover, semaphores and functions such as yield, sleep, suspend and resume are also provided to enable thread control. These primitives are intended for users familiar with threads programming and their implications. For more advanced applications where the execution time may be data dependent, such as compression algorithms, proper selection of the yield points will be important, or else the system may prevent other threads from execution and decrease system reliability. One standard solution is to use timer watchdogs to regain control from misbehaving threads; another is to enable preemptive multi-threading support.

Fig. 2.3 shows a sample application written in the cooperative threading model in Enix. This application is to sense triaxial acceleration and then wirelessly transmit the sensed data to a receiver with ID 1234. There are three cooperative threads in this program: thread0 first initializes the hardware modules and global variables and then blinks the LED periodically; thread1 senses the data while the sensor is ready and the previous sensed data has already been transmitted; thread2 transmits the sensed data to the remote sensor node and then clears the global flag to enable the

next sensing task in thread1. The main function adds the threads to the Enix kernel and then calls enix_kernel_start to invoke the scheduler, which never returns to main().

Priority-Based and Round-Robin Schedulers

Enix provides two scheduling policies: priority-based scheduler and round-robin scheduler to determine the next thread to run based on the thread's priority or registration sequence, respectively. The scheduling policy in Enix is replaceable to provide flexibility for development of WSN applications.

The list of runnable threads is represented as an array of bitmaps to enable quickly finding the next runnable thread. The thread with priority n is runnable only if the n^{th} bit in the bitmap is set; thus, by checking the bitmap, the next running thread can be found. For the priority-based scheduler, the first set bit in the bitmap indicates the next running thread. More powerful ISAs such as ARM support instructions for finding the first set bit in a 32-bit word. Simpler ISAs such as 8051, AVR, or MSP430 do not support such powerful instructions. So, we use a table-lookup implementation. The number of threads is limited by the size of the bitmap.

Algorithm 1 shows the pseudo code for finding the highest-priority runnable thread by table lookup, where *nextPrioTbl* is a byte array with 16 elements, each of which indicates the index of the first set bit in the corresponding element of the array, and *rdylst* is a bitmap list. Each bit represents a cooperative thread with a different priority. By looking up the index of the first set bit, the next running thread can be found as shown in the algorithm.

To implement a round-robin scheduler, we propose another efficient table-lookup algorithm (Algorithm 2). By rotating the *rdylist* to the right for *currentPrio*+1 bits and looking up the table, which is the same as Algorithm 1, the next running thread can be easily found. These two algorithms consume additional 16 bytes of code memory for the immutable table but reduce the total code size and improve the performance significantly, as will be shown in Section 5.5.

Algorithm 1 Fast algorithm to get the next running thread (Priority-Based).

```
if nextPrioTbl[ rdylst & 0x0F ] ≠ 4 then
    return nextPrioTbl[ rdylst & 0x0F ]
else
    return 4 + nextPrioTbl[ rdylst >> 4 ]
end if
```

Algorithm 2 Fast algorithm to get next running thread (Round-Robin).

```
if rdylst = 0 then
    return 7
end if
t ← RIGHTROTATE(rdylst, (currentPrio + 1))
r ← bitmap_lookup( t ) ▷ pass t as rdylist to Algorithm 1
x ← r + currentPrio + 1
if x > 7 then
    return x − 8
else
    return x
end if
```

2.3.2 Compiler-Assisted Virtual Memory

Virtual memory is widely used in OSs to support a larger memory space than provided by the physical memory. In this section, we describe the implementation details of compiler-assisted virtual memory in Enix.

Demand Segmentation

Virtual memory is achieved in Enix via demand segmentation without any hardware support. The code memory of the wireless sensor platform is divided into swappable and non-swappable areas. The Enix kernel and the user-defined logical structures such as the main function are non-swappable. The swappable area utilizes the rest of code memory managed by the virtual memory manager of Enix.

To reduce the runtime overhead in Enix, a library called ELIB is proposed. ELIB consists of binary code segments that are preprocessed on the host PC and is pre-installed on the Micro-SD card, the

secondary storage medium natively supported in Enix. Each segment in ELIB represents the binary code of a function in a position-independent way. That is, a unique virtual address is assigned to each code segment to indicate its location in the Micro-SD card. Calls to ELIB functions from the user program will be translated at compile time into calls to a special run-time loader routine in Enix kernel using a technique called *source code refinement*, to be described in the next section. Therefore, the demanded segments will be loaded into code memory and executed at run-time. The current memory allocation scheme in Enix is first-fit.

It takes three passes to construct ELIB. In the first pass, the common functions are collected into a file named `ELIB.LIB`, which is passed to the library parser to get the binary code size of each function; and then a *virtual-address allocator* is called to allocate a unique virtual address to each function. The second pass is *code modification*: a library function may call another library function that does not exist in code memory, and therefore such code must be modified. The purpose of code modification is to translate ELIB functions to run-time position-independent code. When the code modification is done, the ELIB is compiled and linked to create the file named `ELIB.HEX`, the hex image that consists of the HEX representation of ELIB functions. The final pass splits `ELIB.HEX` into separated HEX files, each of which is called a *code segment*, that is, the HEX representation of a function. Next, the EcoFS installer program installs the code segments onto the Micro-SD card according to their virtual addresses. The procedure above can be done automatically by a shell script. After this procedure, the Micro-SD card is available for loading and executing.

Memory Compaction and Garbage Collection

All virtual-memory systems have the common problem of *fragmentation*. *External fragmentation* occurs when the remaining free memory blocks are not consecutive. Since the dynamic loading library ELIB is runtime position-independent, fragmentation can be solved by memory compaction. A garbage collector routine executes periodically to observe the memory usage and compact memory if necessary.

Another problem arises when the garbage collector reclaims those segments that will be executed after the return of the current running segment. This is called the *cascaded call* problem: it occurs when the code memory is out of use, and the caller is garbage-collected while the callee is executing, such that the callee returns to the caller that has been evicted, thereby causing the system to crash. There are some solutions to fix the cascaded call problem. For example, additional checking code can be inserted before callee returns, and thus the absent caller would be reloaded back before it is returned to. Enix solves this problem by restricting the garbage collector to only non-swappable code.

2.3.3 Dynamic Loading and Run-time Reprogramming

Run-time reprogramming and dynamic loading are important issues in WSN OS design. Deployed sensor nodes need remote reprogrammability for the purpose of bug fixing and firmware updating. Enix supports run-time reprogramming and dynamic loading, and user programs can be updated through RF.

Run-time Position-Independent Code

To achieve fast runtime loading, Enix uses Position-Independent Code (PIC). Traditional PIC is machine dependent, and thus not every architecture supports PIC. We propose an efficient way to generate PIC code without hardware support. With the assistance of run-time kernel via code modification, the code becomes position-independent at run-time. This modification is also applied to the function segments in ELIB as mentioned before. Fig. 2.4 shows the code modification details in Enix. Three types of code are position dependent in general Enix user programs.

First, kernel calls in the user program do not have to be modified, due to the separation between the kernel and user programs. The linker redirects these kernel calls to the appropriate addresses

by a linker script as shown in Fig. 2.4(a).

The second type of position-independent code covers the library calls as shown in Fig. 2.4(b). We redirect this kind of absolute calls to the special kernel function via code modification. The kernel function finds the address of target function at run-time and then jumps there. If the target does not exist, then the loader is invoked to do the same work as mentioned in the virtual memory section.

The last type is for local absolute jumps, as shown in Fig. 2.4(c). In most cases, local jumps are relative jumps except for those jumping from the beginning of a large logical structure to the end. We convert a long jump instruction into a relative jump routine by calculating the relative size from the source to the target at compile time, get the current program counter at run-time, and then add the program counter and the relative size to get the target address at run-time.

In addition, Fig. 2.4(d) also shows two key functions in the Enix kernel to support run-time PIC. The function enix_getpc gets the program counter at run-time and stores it into the *DPH1*:*DPL1* register pair (an alternative pair of the Data Pointer High/Low registers in the 8051 ISA). Since the *lcall* instruction will push the return address onto the stack, we can call enix_getpc to read the program counter from the stack. enix_fake is another kernel function that checks the existence of the target function and redirects the *lcall* instruction to the target function address at runtime. If the target function does not exist in the code memory, then it will be loaded from the Micro-SD card according to the virtual address passed into enix_fake function.

Source Code Refinement

In order to hide the details of the run-time loader from the flow of user program development and to reduce the amount of code modifications, a source code refinement technique is applied. Fig. 2.5(a) shows a simple Enix user application sending a string of data through UART. For the include files of Enix user program, the function definitions are removed, and C macros are applied as shown in the Fig. 2.5(a). By using macros and *varargs.h* support for the C language,

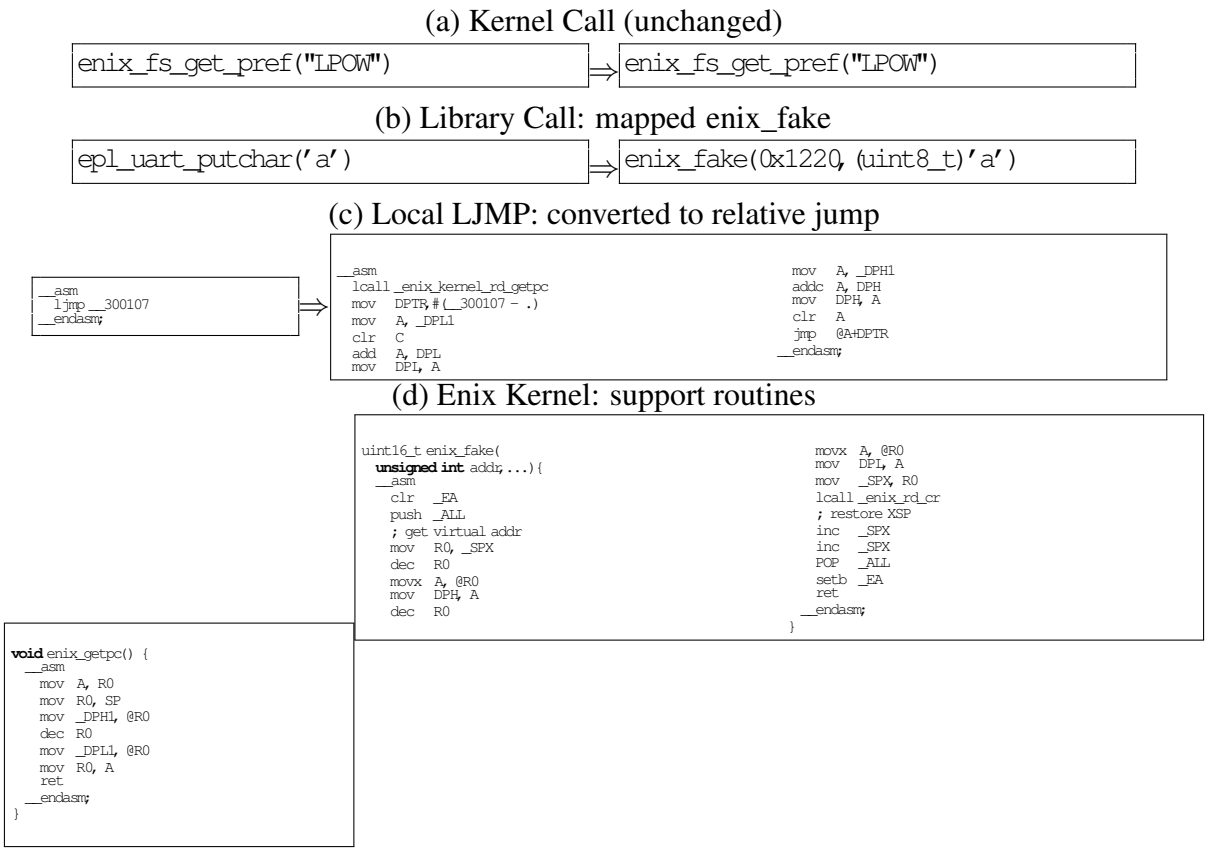


Figure 2.4: (a)-(c): Transformation and (d) support for position independent code.

User Program

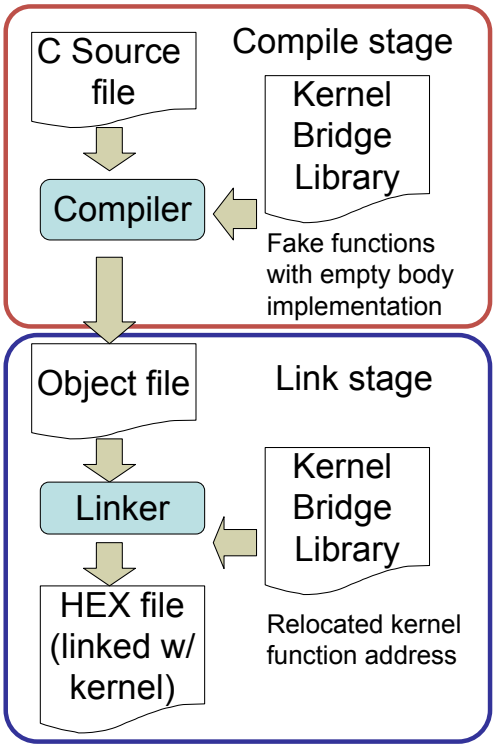
```
extern xdata char* malloc_ptr_1;
int main() {
    LED0 = LED1 = OFF;
    epl_uart_init(UART_BAUD_57K6);
    epl_uart_putstr("test_uart\n");
    return 0;
}
```

Source Code Refinement

```
extern xdata char* malloc_ptr_1;
#define _EPL_UART_PUTSINT 11776 //size:145
#define _EPL_UART_PUTSTR  14336 //size:110
#define _EPL_UART_INIT    14848 //size:95

#define epl_uart_init(a) \
        enix_fake(_EPL_UART_INIT, \
         (unsigned char)a)
#define epl_uart_putchar(a) \
        enix_fake(_EPL_UART_PUTCHAR, \
                 (unsigned char)a)
#define epl_uart_putstr(a) \
        enix_fake(_EPL_UART_PUTSTR, \
                 (unsigned char*)a)
```

(a) Source code refinement.



(b) Bridge library.

Figure 2.5: Bridge library and Enix user programs.

each library function call in the user program is redirected to a special function named enix_fake. This vararg-style function allows variable numbers and types of parameters, and therefore every function prototype can work with this refinement with additional type casting. Furthermore, the source code refinement technique can be used to achieve system configuration. For example, a general library function SendPacket can be called by the user program to send a byte buffer through different interfaces, such as RF, UART, I^2C and SPI. It is easy to provide a configuration interface for users to choose the transmission interface of SendPacket library function by using the source code refinement technique. Hence, different hardware modules of wireless sensor devices can be easily configured.

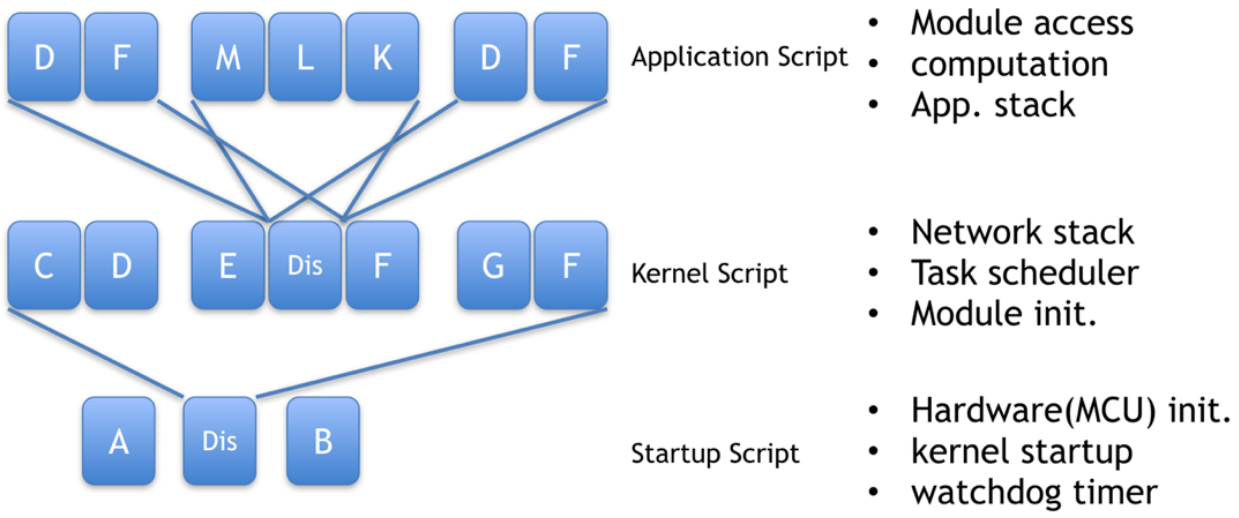


Figure 2.6: Dispatcher on top on a existing schedulor.

2.4 From Enix to Current Proprietary Schedulers

While modern node devices such as EcoBT, EcoBTmini are providing industrial standard wireless communication protocol, BLE. It is inevitable to use vendor provided proprietary scheduler in order to use the integrated protocol stack.

We propose a way to integrate a dispatcher on top of proprietary scheduler by using threaded code concepts. A task is composed by multiple actors. An actor is an executable code snippet that can be execute in the existing schedulor. For example, in CC2541, TI has provided a OSAL schedulor, a software event driven schedulor. This threaded code dispatcher can be integrated as shown in Figure. 2.6. In the executiong loop, each time a dispatcher event is triggered, an actor is executed and finished parts of a task. After several triggered events, a task is completed. This can avoid a long execution task jeoparize underlying scheduler with running protocol stack.

An actor can be a function exists in the firmware image, or a newly uploaded code snippest and executed from ram. We also design two characteristics for memory access and function execution in EcoBT. To access to memory blocks in EcoBT, developers can use address and length to get the content of the requested memory block. The format is shown in Figure. 2.7.

Request

Operand	Idx 0	1 - 4	5	6	7
Create	0x01	ID(4-byte)	Storage Class	Type	Block Length
Read	0x02	ID(4-byte)	Offset	Block Length	
Update	0x03	ID(4-byte)	Offset	Block Length	Data
Delete	0x04	ID(4-byte)			
List	0x05				

Response (through notification)

Operand	Idx 0	1 - 4	5	6	7
Create	0x01	ID(4-byte)	Length of Addr	Addrress	
Update	0x03	ID(4-byte)	Offset	Block Length	Data
List	0x05	ID(4-byte)	Storage Class	Type	Block Length

Figure 2.7: Characteristic content format for memory operations.

Upon receiving requests in EcoBT, an actor load the content from the memory block and save in ram area that can be diliver to the user in the next actor. As for executing functions on EcoBT, if a user try to excute an existing function in EcoBT, a tool is used to parse the source code and compiler generated map to get target address in flash. With the target function address, the user can request EcoBT to execute the function with the function execution characteristic in Figure. 2.8. User can also upload a compiled code in hex format into EcoBT and execute.

With memory accessing and function execution capabilities, We can have the EcoBT works like the EcoSpire with EcoCast.

With the PyBLE description in 3.6, we build a EcoBT wrapper. With the wrppaer, we can directly access to EcoBT's internal memory as shown in Figure. 2.9.

In current implementation, not only the data memory can be accessed. The SFRs such as Px, PxDIR can be access and modified over the memory BLE characterisitc. A user can do simple GPIO control over this and achieve some automation control. This also equipts the user to develop

Request

Operand	Value	1 - 4	5	6 - 7	8	9 -
Create	0x01	ID(4-byte)	Storage Class	Blob Size(2-byte)	Return type	Param Num
Extend	0x02	ID(4-byte)	Param offset	Param Type	...	
Upload	0x03	ID(4-byte)	Offset	Data		
Prepare	0x04	Session(4-byte)	ID(4-byte)	Offset	Param	...
Execute	0x05	Session(4-byte)				

func type: ram, flash return/param type: UINT8, INT8, UINT16, INT16, UINT32, INT32, PTR tag: a 32-bit long tag that can be used for git hash, tag[3], tag[2], tag[1], tag[0]

Response

Operand	Value	-	-	-	-
Create	0x01	ID(4-byte)	Length of Addr	Addr	
Return	0x02	Session(4-byte)	data	...	

Figure 2.8: Characteristic content format for function operations.

```
mcu = pymcu.CC2541(comm="ble")
print mcu.P0
print mcu.P1
print mcu.P2
print mcu.P0DIR
mcu.P0DIR = 0xFF
print mcu.P0DIR
mcu.P0 = 0x02
print mcu.P0
mcu.P0 = 0x00
```

Figure 2.9: Access SFR and memory in the MCU.

```
# LED on/off in ram code
mcu.func_alloc("ram_on", storageClass='xdata', body=bytearray([0xC2, 0x85, 0x22]))
mcu.func_alloc("ram_off", storageClass='xdata', body=bytearray([0xD2, 0x85, 0x22]))
while True:
    mcu.func_exec("ram_on", args={})
    time.sleep(1)
    mcu.func_exec("ram_off", args={})
    time.sleep(1)
```

Figure 2.10: Ram code: Upload and execute.

```
# Existing Function Call Execution
mcu.func_alloc("led0_on", hash_id=imageMap['LED0_on']['address'], storageClass='code', retType=None, args=None)
mcu.func_alloc("led0_off", hash_id=imageMap['LED0_off']['address'], storageClass='code', retType=None, args=None)
mcu.func_alloc("led1_on", hash_id=imageMap['LED1_on']['address'], storageClass='code', retType=None, args=None)
mcu.func_alloc("led1_off", hash_id=imageMap['LED1_off']['address'], storageClass='code', retType=None, args=None)
while True:
    mcu.func_exec("led0_on", args={})
    time.sleep(1)
    mcu.func_exec("led0_off", args={})
    time.sleep(1)
```

Figure 2.11: Calling existing function calls.

a tool that can inspect the state of EcoBT. Before, a developer can only insepct the state of the MCU with directly connected programmers. Now we can use this memory accessing capability to do primitive edge device monitoring.

For the function execution capability, user can have more complicated operations executed on EcoBT. In Figure. 2.10 and Figure. 2.11, we show how to execute ram code and existing functions in the image.

We use the threaded code concept of actors and use it to implement a dispatcher on top on a existing scheduler in EcoBT. In this way, we can still utilize vendored provided BLE protocol stack and build a application to allow flexibility to access memory and execute functions on EcoBT.

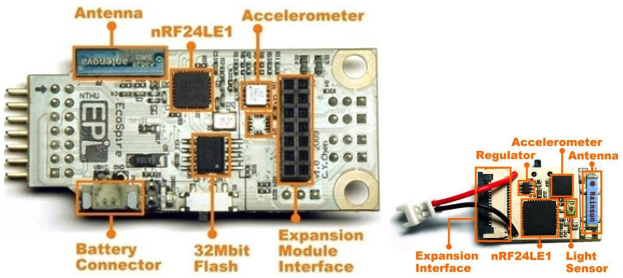


Figure 2.12: (a) EcoSpire node: 23×50 mm^2; (b) EcoSpire simple node, 13×20 mm^2.

2.5 Evaluation and Results

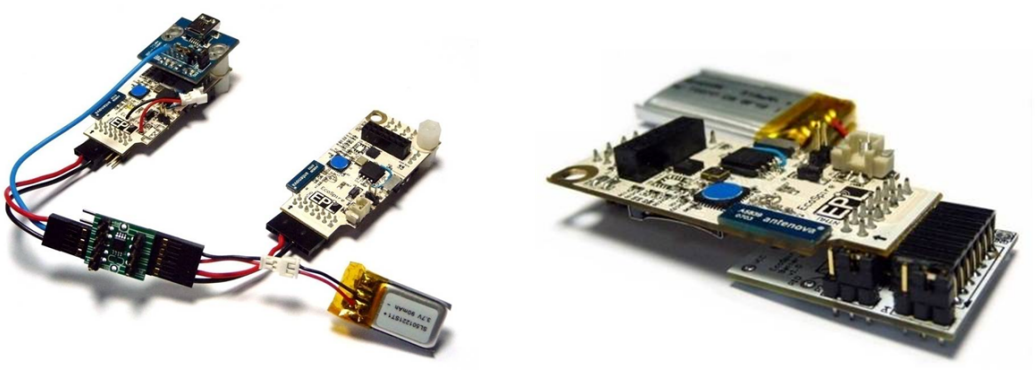
This section shows the efficiency of Enix as a lightweight dynamic WSN OS. We first describe our experimental setup, including our hardware platform and software tools. Second, we compare the context switch overhead and the code and data memory footprints of different multi-threaded scheduler implementations to show that the cooperative threads model of Enix has the lowest overhead. We also compare the power consumption of loading code segments over RF and Micro-SD cards. We evaluate our code updating scheme against other methods to show that we reduce runtime overhead significantly in terms of the size of the uploaded code.

2.5.1 Experimental Setup

This section describes the hardware platform and software tools for Enix.

Sensor Platform

EcoSpire [12] is our experimental platform. As shown in Fig. 2.12, EcoSpire refers to the larger version for prototyping and application development, and a compact version called the "simple node" is more suitable for field deployment. This platform consists of an MCU with an integrated RF transceiver, a chip antenna, a triaxial accelerometer, power circuitry, and an expansion inter-



(a) Current sensor. (b) Battery fuel gauge.

Figure 2.13: Power measurement modules.

face. The MCU core runs at 16 MHz by default and comes with 16KB program flash and 1KB data RAM. EcoSpire's nRF24LE1 RF-MCU contains an on-chip radio that implements the Enhanced ShockBurst protocol, which is the core of the newly finalized Bluetooth 4.0 Low Energy Technology standard. A 32-Mbit on-board flash memory and Micro-SD expansion capability are included for nonvolatile data storage. This configuration is actually quite representative of many integrated RF-MCUs made by TI (CC2430, CC2510) and Z-Wave ones, all with integrated 8051 cores, and they comprise well over 90% of the RF-MCUs on the market. Most WSN OS cannot fit into this limited platform and are thus not easily comparable.

To measure the energy consumption of Enix, we developed two power measurement modules as shown in Fig. 2.13. One is the *current sensor module* for measuring the instantaneous current of another EcoSpire in execution. The other module is the *battery fuel gauge*, which allows EcoSpire to measure the power and battery capacity itself at run-time.

Software Tools

The software for EcoSpire includes an IDE and graphical user interface tools (GUI) on the host computer, system software on the sensor node and the base station, and utility tools for image uploading and RF debugging. We built our IDE with Eclipse by creating a plugin for EcoSpire

Table 2.3: Context switch overhead comparison between algorithms for selecting the next running thread. Unit: μs.

	RR	Priority-Based
Conventional Linear Check	48.925	55.2
Our Fast Table Lookup	16.325	11.9

Table 2.4: Context switch overhead comparison between different scheduler implementation. Unit: μs.

	RR	Priority-Based
C-Coroutines	16.325	11.9
ASM-Coroutines	8.25	8.475
Preemptive	13.45	23.3

development. With this plugin, we provide a fully GUI-based programming environment. This lowers the burden for programmers to memorize commands and enables them to focus on the software development process, from editing, compiling, and linking to firmware programming.

2.5.2 Context Switch Overhead of Different Schedulers

To evaluate the context-switch overhead of our cooperative threads in Enix, we implement both round-robin (RR) and priority-based schedulers with different algorithms that may affect the context-switching time.

Table 2.3 shows the results of using our fast table-lookup algorithms versus straightforward linear search to find the next running thread from the runnable queue. We measure the execution time by taking the average of 60,000 context switches. It is clear to observe that our fast algorithms significantly improve both RR and priority-based schedulers by cutting the execution times down to a quarter of the original linear search implementation.

Table 2.4 shows a comparison of context-switch overhead between different multi-threaded models for both RR and priority-based scheduler. Preemptive multi-threading has the highest context-switch overhead due to the unpredictable preemption time, and therefore all of the registers must

Table 2.5: Context switch overhead comparison between Enix and μC/OS-II by averaging 60,000 context switches.

OS	Enix		μC/OS-II
policy	Coop.-th.	Preemp.-th.	default
Overhead (μs)	8.47	23.3	250

Table 2.6: Code and data size comparison between different scheduler implementation. Unit: bytes

Model	C-Coroutines		Cooperative		Preemptive	
Scheduler	RR	Prio.	RR	Prio.	RR	Prio.
Code Size	918	897	1140	1051	1688	1559
Data Size	79	79	70	69	97	91

be saved and restored during context switch. We implement C-coroutines using C-switch statements and add the priority-based and RR schedulers to it. C-coroutines and cooperative threads have the feature that context switching occurs only when the running thread calls yield or sleep functions, and thus they have lower context-switch overhead. The implementation of C-coroutines uses C-switch statements, and this means that every context switch results in several comparisons of variables and an absolute jump. Consequently, a context switch of cooperative threads is simply a replacement of the program counter, stack pointer, and some global variables, and thus it has the lowest overhead.

To compare Enix with a real-world OS, we ported μC/OS-II to EcoSpire. It is widely used in industry, whereas no other WSN OSs are known to run on the 8051. Table 2.5 compares the context-switch overhead of μC/OS-II and our work. The reason why μC/OS-II has high context-switch overhead is that it uses external memory to store both the per-thread stack and registers, thus incurring great overhead from many external memory movements.

Table 2.6 shows a comparison of the code and data memory usage between different scheduler implementations. The preemptive ones consume the most memory for the same reason mentioned before. Other scheduler implementations require about 1KB of code memory, which is frugal compared to other regular RTOSs such as μC/OS-II and FreeRTOS, as shown in Table 2.7.

Table 2.7: Code and data size comparison between Enix, FreeRTOS and μC/OS-II. Unit: bytes

	Enix(Scheduler)	FreeRTOS	μC/OS-II
Code Size	918	7560	10294
Data Size	79	719	488

2.5.3 Virtual Memory and EcoFS

This section shows the speed and power consumption of SD cards and other serial flash memories on EcoSpire. The results confirm the reason that the Micro-SD card was chosen for the main nonvolatile storage, as discussed in Section **??**.

Tables 2.8 and 2.9 compare the speed and power consumption of a Micro-SD card with three other different on-board serial flash memories. These flash memories and the Micro-SD card are connected to EcoSpire through a common SPI bus. The SD card has fast sequential read and sequential write properties but poor random access speed due to the characteristics of NAND flash memory used by the SD card. Most of the routines in EcoFS use sequential reads and sequential writes such as code data and sensed data. For the other preferences types and network data, their access unit is a sector, and therefore the access time is equivalent to sequential access. Thus, the slow random access speed does not affect EcoFS. For the power consumption, the on-card controller of the SD card causes the highest power consumption among all flash memories. In fact, the data shown in Table 2.9 is the active power consumption of the SD card, that is, when the chip-select ($\overline{\text{CS}}$) signal is asserted. When $\overline{\text{CS}}$ is de-asserted, the power consumption of the SD card is low. Accordingly, the appropriate usage of the SD card may reduce the total energy cost of sensor nodes. EcoFS is designed such that once the SD card is selected by $\overline{\text{CS}}$, it will finish the I/O operations as soon as possible.

Fig. 2.14 shows the time and energy cost of the sensor node performing 1MB of sequential read with different block sizes. Due to the requirement of a start command before each sequential read and sequential write operation, the highest performance and lowest energy cost happen while the maximum block size of 512 bytes is applied. EcoFS tries to use the largest possible block size

Table 2.8: I/O speed comparison between flash chips. Units: Kbytes/s.

	SST25 VF512A	SST25 VF032B	Pm25 LV020	SanDisk MicroSD
Sequential read	170.7	170.7	176.6	170.66
Sequential write	46.5	71.1	54.8	92.75
Random read	46.4	39.4	43	2.18
Random write	20.9	23.1	17.3	0.11

Table 2.9: The energy consumption comparison between different flash chips. unit: mJ.

	SST25 VF512A	SST25 VF032B	Pm25 LV020	SanDisk MicroSD
Energy Write 1MB	397.866	501.564	481.404	976.794
Energy Read 1MB	146.454	103.656	85.638	517.314

and reduce the number of random-access operations in order to overcome the power and speed bottlenecks of the SD card. Owing to the different approaches by SD card manufacturers, we have tried seven 2GB SD cards made by different manufacturers. We measure the speed and power consumption of sequential read and sequential write operations shown in Table 2.10. This comparison enables the user to make price/performance trade-offs.

We also evaluated the delay and energy of the virtual memory. They are 6.328ms / KB and 505.2μJ / KB, respectively.

Table 2.10: SD card comparison between manufacturers.

	Speed (KB/s)		Power Consumption (mW)	
	Seq. Read	Seq. Write	Seq. Read	Seq. Write
Toshiba	170.7	51.2	133.9	82.8
Kingston	170.7	37.9	145.5	76.7
SanDisk	170.7	92.8	81.4	86.2
TOPRAM	113.8	51.2	150.1	121
Team	73.1	73.1	105.9	108
Silicon Power	128	53.9	154	118.8
Transcend	93.1	92.8	98.2	117.8

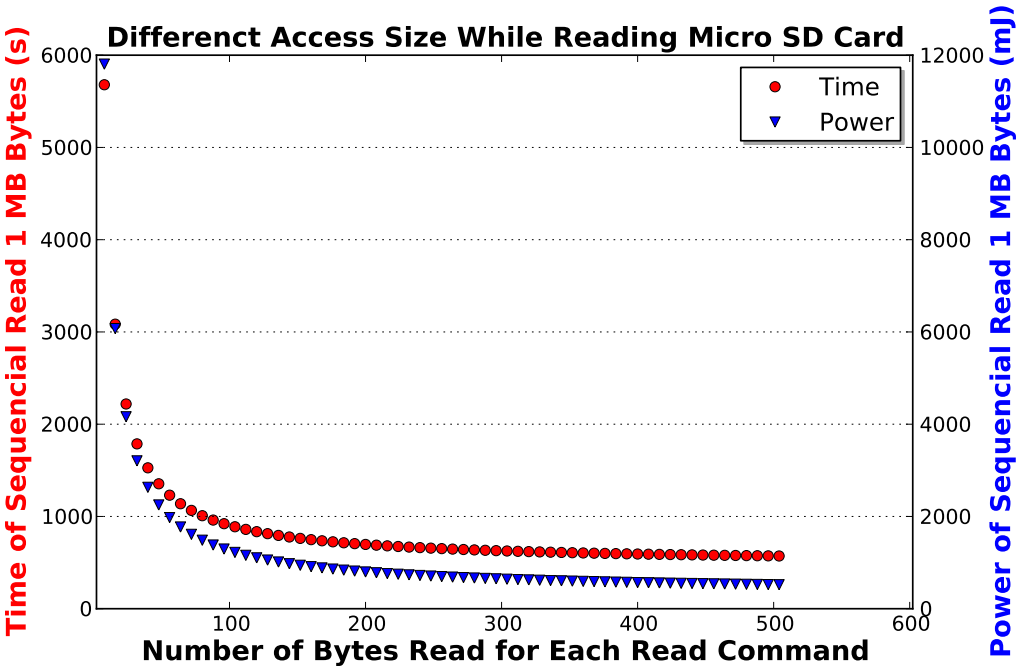


Figure 2.14: SD card reading using different block lengths.

2.5.4 Efficiency of Enix Code Update Scheme

The key concept of Enix is the separation of user-defined logical structures and commonly used library functions. By storing the dynamic loading library ELIB on the Micro-SD card, only the user-defined logical structures need to be remotely programmed through the RF. Hence, the number of RF packets for run-time reprogramming is reduced.

We develop five WSN applications compiled with and without Enix. The first three applications are general WSN tasks:

(1) sense and transmit data to the base station through RF,

(2) sense and log data onto the Micro-SD card,

(3) receive and forward packets to other sensor nodes through RF.

(4) EcoNet Transmit, and

(5) EcoNet Receive.

EcoNet is a simple multi-hop network composed of several EcoSpire sensor nodes. The unique ID and the adjacent nodes are all recorded on the Micro-SD card of each sensor node. Every sensor node can invoke the EcoFS API to enumerate its adjacent sensor nodes. The fourth application, EcoNet Transmit, collects the sensor data with a random number and transmits them to the neighboring sensor nodes by enumerating the network data block of EcoFS. The last application EcoNet Receive receives the sensed data from neighboring sensor nodes and checks the duplication of random numbers of sequential RF packets, and forwards the valid packets to the base station.

Table 2.11 compares the uploaded image sizes of the above five WSN applications with and without Enix. The WSN applications without any OS support have binary image sizes larger than 6KB on average. Due to the large code size of the RF library, only the second application has a code size less than 3KB. By comparing the same applications that use Enix as the OS, the sizes of the program images to be transmitted through RF are reduced significantly. These applications produce 500 bytes of binary image on average except for the fourth application, EcoNet Transmit, which generates random numbers without calling any kernel or ELIB functions, and therefore it produces about 1KB of program image.

Most of the run-time reprogramming schemes use the VCDIFF tool to generate the patch between two binary code images, and the patch will be applied by the sensor node. Table 2.12 shows the results from running VCDIFF for each of the two different WSN applications mentioned above. Although the average binary image size is reduced to 4KB, it is still larger than the applications written on top of Enix.

Table 2.13 compares the energy cost of 1MB data transfer with an SD card and over RF. As the table shows, reliable RF transmission and reception consumes the highest energy. Thus, for code swapping purposes, it is more energy efficient to swap code from an SD card on demand than to swap over RF. In short, by applying Enix as the operating system, sensor nodes can save energy

Table 2.11: Runtime reprogramming code size with/without Enix. Unit: bytes.

application binary size	Sensing & RF Tx	Sensing & Log	RF Tx & RF Rx	EcoNet Tx	EcoNet Rx
no OS	4825	2903	7646	8233	7770
on Enix	474	673	514	1027	442

Table 2.12: Update code size using VCDIFF delta compression. Units: bytes

Xdelta -9	Sens.& RF Tx	Sens. &Log	RF Tx & Rx	EcoNet Tx	EcoNet Rx
Sensing&RF Tx	X	3148	2834	2663	2810
Sensing&Log	2920	X	1937	1921	1936
RF Tx & Rx	4509	4730	X	3168	3226
EcoNet Tx	4927	5201	3760	X	3847
EcoNet Rx	4523	4753	3246	3291	X

and time while becoming capable of efficient remote reprogramming due to the reduced binary image size.

Table 2.13: The energy consumption comparison between SD card and RF (Full Speed). Units: mJ.

	Reliable RF	RF	SD Card
Read/Rx 1MB	1540	1306	517
Write/Tx 1MB	1342	450	977

Chapter 3

EcoCast: An Interactive Framework for Parallel Execution of Wireless Sensor Nodes in Multi-Hop Networks

3.1 Related Work

3.1.1 Remote Firmware Update

Code swapping over the air can be considered a special form of remote firmware update, which means modifying the firmware image over a wireless communication link. The techniques can be divided into full vs. partial image replacement. One may completely replace old binary image with a new one and reboot, and this has been done for single-hop [15] and multi-hop [80, 50] networks, including epidemic protocol [33] for rapid dissemination. However, full image replacement is costly, and partial image replacement techniques have been proposed, by transmitting a *diff script* or an *edit script* containing patching commands. Proposed techniques use diff [76], block-level

Table 3.1: Comparison of partial image replacement techniques.

work	platform	data	code	approach
[76]	EYES	2KB	60KB	diff-based edit script
[39]	Mica2	4KB	128KB	Rsync algorithm diff
[48]	Mica2	4KB	128KB	remote linking / deltas
this	nRF24LE1	1KB	16KB	host-assisted loading

comparison [39], by finding fixed-size shared blocks [83], or by leveraging higher-level program structure to generate more concise scripts (called *deltas*) that exploit the program structure to generate more concise scripts [48]. However, even a small script can lead to much memory perturbation due to shifting memory content, leading to wear and tear in addition to high energy consumption [42]. Moreover, these are not interactive techniques, and they require rebooting after a new program image has been installed or patched. Program states will be lost under such circumstances and thus they are not suitable for interactive style of execution. Moreover, there is no bound on when the reprogramming will be completed, and there is no confirmation back to the host at all.

An alternative to program image replacement is to use loadable modules for reprogramming. Loading a module entails resolving references to symbols and can be divided into *pre-linking* [20] and *dynamic linking* [19, 58] *dynamic loading* [11, 27]. Pre-linked modules are smaller in size and require less information to be transmitted and less complex to load, while dynamic linking requires less effort to maintain program image consistency in case of physical address conflicts. Mantis OS (MOS) [**?**] implements a lightweight POSIX threads and supports remote reprogramming by implementing a command server on the node. Remote reprogramming is limited to modify constants via the command server, but general function replacement, are not supported.

A shell is an interactive command-line interpreter for a scripting language, while a scripting language is a "gluing" language whose primitives consist of higher-level functions or programs rather than low-level instructions. Scripting languages need not be executed interactively but may be executed in batch. They can be divided into systems whose interpreter runs on either the node or on the host computer.

Table 3.2: Comparison of scripting systems for WSN.

Scripting System	Interpreter resides on	Turing Complete	Weight
SensorWare Tcl[64]	node	yes	very heavy
TinyScript[54]	node	yes	above Maté+TinyOS
LiteOS shell[10]	node+host	no	medium
Tapper[86]	node	no	light
Rappit[26]	host	no	light
EcoExec[31]	host	yes	light

Among systems that run the shell on the node, SensorWare [8] handles a subset of the Tcl [64] language but requires essentially a PDA-class system, too costly for most sensor nodes. Several other systems also support shell on the nodes, including Mantis, Contiki, and TinyOS. These are useful for users to invoke commands interactively, but they occupy a significant part of the program and data memory. To fit in very small memory, Tapper [86] is a lightweight scripting engine that is optimized for parsing predefined keywords and values in a simple syntax and invoking the corresponding primitives. While useful for simple command invocation, it does not support general programming. BerthaOS running on the PushPin wireless sensor nodes can handle Pfrags [55], which are limited-sized (2KB) program fragments that can be dynamically loaded and executed to achieve interactivity. However, the host-side interface appears more limited.

3.1.2 Shell and Scripting Systems

One solution to overcome the resource limitation on the nodes is to run the shell on the host computer, rather than on the node. LiteOS [10] provides LiteShell, a command-line interpreter that supports unix-like commands and file system naming convention for invoking commands and updating firmware on the nodes in reachable WSNs. The commands are also available as APIs for developing user applications on the host PC. EcoExec [31] also runs a wireless shell on the PC, except it uses Python, a dynamically typed, object-oriented scripting language instead of a unix-like CLI. Moreover, it also involves the compiler in the loop by dynamically generating optimized

code fragments and swapping code to the nodes on demand, although the user needs to write it in C. For interacting with multiple nodes, EcoExec relies on either iterative constructs (for-loop, while-loop etc) or using Python's built-in functional programming constructs (map, reduce, filter) but these would be executed serially. The runtime scales linearly, and this is considered inefficient as much of the communication and execution can be parallelized or pipelined. Another issue is synchrony: nodes may take a different amount of time to finish executing a function, or the results may be invoked at different times due to the different number of hops that the packets may need to travel. Whether it is necessary to synchronize the nodes before the next command is allowed to proceed becomes an important consideration in such a parallel language. EcoExec with its serial semantics by default assumes synchronous (fully blocking) semantics, which is the most conservative and at the same time slowest.

3.1.3 Routing protocols in WSNs

Routing protocols can be classified by how the source node finds a path to the destination node: proactive, reactive, hybrid, and hierarchical. Proactive protocols [70, 14, 63, 61] need to compute routes in advance, while reactive protocols [71, 41] compute routes on demand. The former incurs more overhead in ensuring updating and maintaining routing tables and may be slower to respond to changes in the network, but once the routes are established, then communication can be efficient. The latter is more adaptive to changes in the network (e.g., node failure, moving) but pays high cost in terms of packet flooding. Hybrid schemes are possible by starting with proactive routes and then reactive flooding on demand, by localizing the control messages to a small set of nodes near the change [69, 3]. However, a hybrid scheme is not necessarily more scalable, since more packets can be dropped when the number of sources increases [9]. Hierarchical routing schemes basically rely on local cluster heads or nodes of another tier to more evenly distribute the work [40, 28].

In this paper, we present a multi-hop scheme that is similar to proactive routing protocols, but

nodes do not maintain any routing table since topology information is maintained by the Network Manager that resides on the host side. We also incorporate reactive routing because route are computed on demand.

3.1.4 Query systems

Query-like interfaces have been proposed for WSN [7, 78, 88], including an SQL-like interface [57] for retrieving data from the network of nodes. The query syntax follows that of the SQL language and is fully interactive. However, they are not for supporting reprogramming. In our work, we not only can query data from nodes but also can upload programs in native code.

3.2 System Overview

EcoCast refers to the software architecture that ties together a wireless sensing platform. This section describes a specific platform that we used to demonstrate the feasibility of the proposed idea, plus how the different subsystems interact with each other.

3.2.1 Wireless Sensing Platform

The three main subsystems the wireless sensing platform that we use are the *nodes*, *base stations*, and the *host*. An overview of these subsystems is shown in Fig. 3.1.

Nodes

Unlike what the name suggests, EcoCast actually does *not* currently use the Eco ultra-compact wireless sensor nodes. Instead, it uses a compatible platform based on the Nordic nRF24LE1

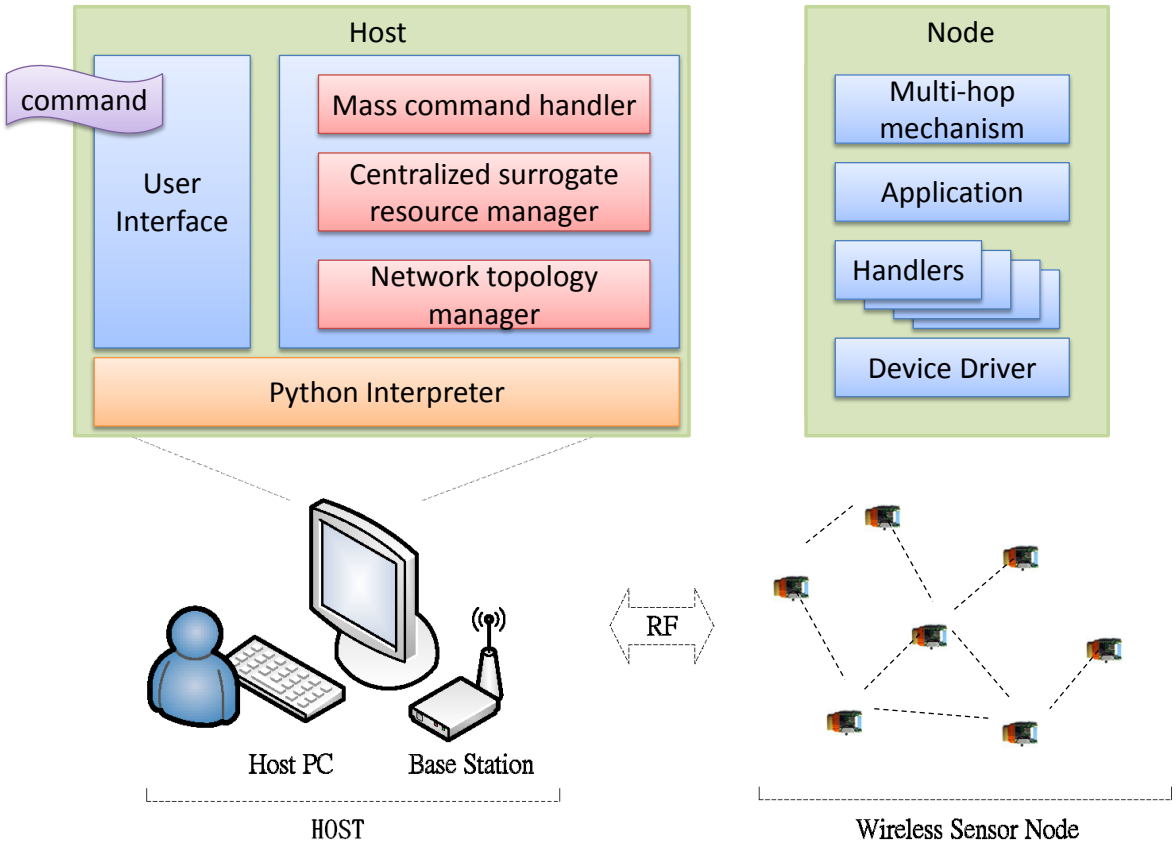


Figure 3.1: System overview.

MCU, which contains an 8051-compatible core, the nRF24L01 radio in the 2.4 GHz ISM band, a multi-channel analog-to-digital converter, and general-purpose I/O pins. The hardware supports reliable communication, including auto-ack and auto-retransmit. This is one of the smallest commercially available platform to date, available in 4×4 to 6×6 mm^2 packages with 1 KB of data RAM and 16 KB of flash. Other platforms in this class include the TI CC2510F16 with an IEEE 802.15.4-compliant radio instead. The nodes are assumed to perform sensing, actuation, relaying, or any other function just as any embedded system.

Base Station

We chose a base station with Fast Ethernet for its ability to support high-speed transfer and networking over TCP/IP, although EcoCast does not depend on any specific interface. The base station hardware here is built by connecting a Nordic nRF24L01 2.4 GHz RF transceiver module to a Freescale DEMO9S12NE64 evaluation board over its 40-pins expansion port. It is programmed to relay packets between Ethernet and the RF side.

Host Computer

The host subsystem runs on a conventional PC that can communicate with the nodes via the base station(s). The host computer runs the wireless shell that enables programmers to interact with the nodes over the air. The host also maintains a suite of tools that help the shell keep track of the state of the nodes as well as code generation and optimization. The tools include the compiler, linker, runtime estimator, node database, and version control. These tools are invoked automatically and transparently to the user as the type commands into the shell. The execution sequence is explained next.

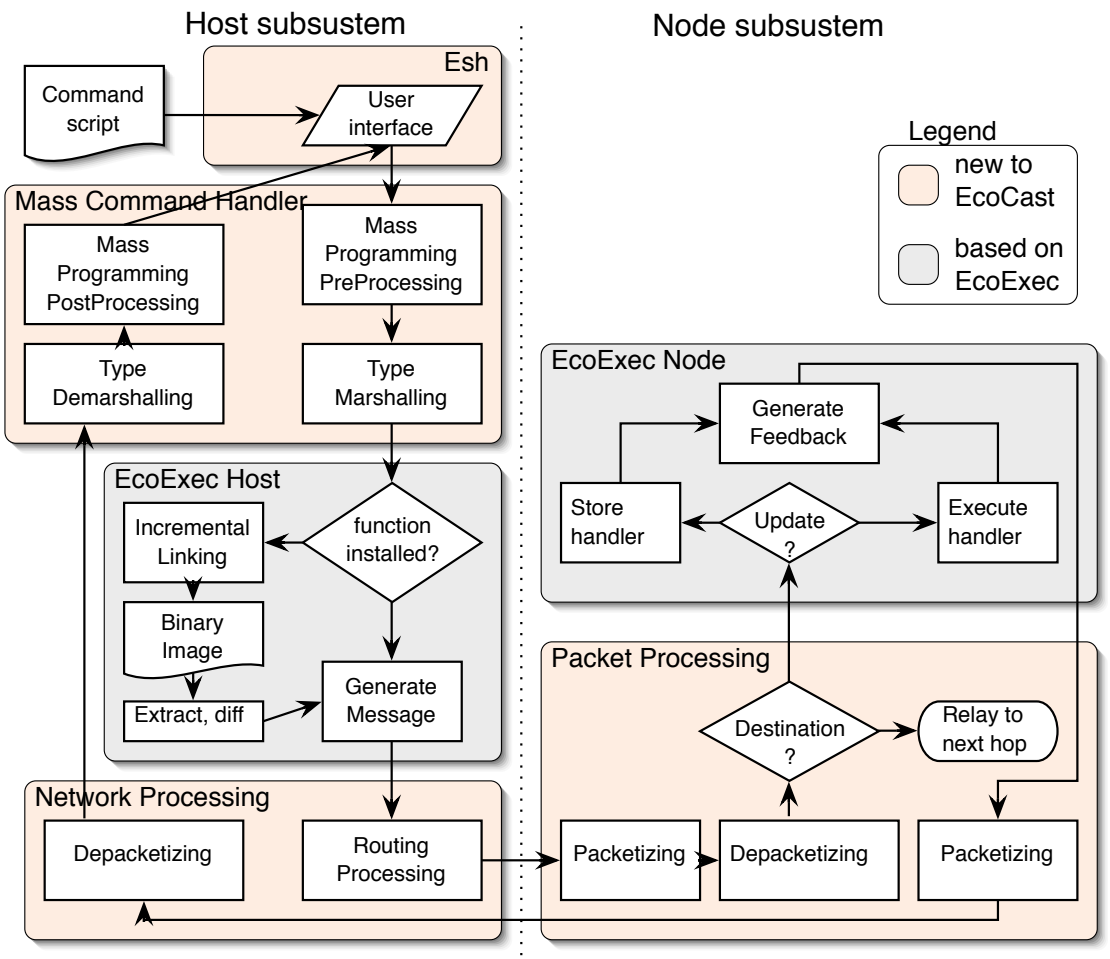


Figure 3.2: Execution Flow of EcoCast.

3.2.2 Execution Flow

The user can either type in a command interactively or run an application program (script) written in Python to call our API library. In either case, the same command is invoked, and EcoCast handles them in exactly the same way. The steps in the execution are node handle creation, command processing, and message issuing. The flowchart of EcoCast is shown in Fig. 3.2.

Joining and Leaving a Network

When the host system starts up, it first searches for available sensor nodes to join the network. We assume every node has a unique id out of factory. On power-up, a node which has not been bound yet tries to bind to a host system if there is any or it waits for the host to do pairing while a bound node tries to find the host it used to bind to before it preforms the same procedure. A bound node is also capable to pair to a new host system actively. Once a node and a host agree to connect, the host constructs a *node handle* object in Python with the unique ID of node and creates a new entry for it in the node database. Subsequently, all operations on the node, including function invocation, code update, configuration changes, etc, are all done through the node handle object. This idea is analogous to the concept of file handles used by most programming languages for reading and writing files, except these calls translate into wireless communication messages between the host and the node. A node is leaving the network if the host cannot access it and the host system removes the relevant node handle object and information in database. A timeout based method is used to decide when to delete a node.

Language Wrapping

Similar to other language wrappers for middleware systems, our wrapper also performs *type marshalling* and *demarshalling* when translating between Python code and the node's native code. This requires that the host side know about the architecture profile of the target MCU (endian, word size, etc) and interface to the function prototype (in C), including the types of the parameters and return value. EcoCast handles the checking and conversion, by raising exceptions if necessary.

Demand Paging and Dynamic Compilation

EcoCast is much more than just a language wrapper for function invocation and type marshalling. It also performs many OS functions on the host on behalf of the nodes, the most important of which is host-assisted demand paging. Dynamic compilation is also performed as a step if necessary. Before invoking a function, EcoCast checks the state of the node's firmware image to see if the target function is in memory. If not, then it attempts to swap in the code first, if the binary exists. If the binary does not exist, then it attempts to compile and link the code from either library source code or even Python code fragments. It needs to perform incremental linking on the new function to produce a new binary image.

Communication and Synchronization

Ultimately, everything the host needs to do to a node is done by sending messages via the base station that the node is associated with. Messages from the host can either specify that new code be installed or a target function be invoked. A single command in Python may turn into commands to multiple nodes and gathering results from their invocations. EcoCast not only attempts to optimize such communication patterns by broadcasting and scheduling, but also supports several different user-specifiable semantics, including fully blocking and nonblocking execution styles.

3.3 Scripting

This section describes the details of interactive execution involving the host-side scripting environment.

3.3.1 Scripting Language

EcoCast provides a scripting environment as the primary way for users and application programs to interact with the sensor nodes. It consists of a class library at a higher level for the user to access the sensor network, a shell called Esh for interactive access, and a class library at a lower level for runtime support.

Python language

We use Python [1] as our scripting environment for its clean syntax and rich feature support. It can be run two ways: interactive mode and batch mode. In interactive mode, users can enter commands as Python statements directly into the prompt on-the-fly. Variables in Python need not be declared before they are used, and they track the references to objects that are self descriptive (and thus considered dynamically typed). Typing the name of a variable in interactive mode causes the object to be "rendered" in its string representation. Thus, in interactive mode,

```
>>> x = 3    # no need to declare x
>>> x        # displays its value as text
3            # interpreter calls int's __repr__ method
>>> x + 2
5
```

The runtime system keeps track of the types of the objects dynamically and enforces their consistent use. One direct advantage is that the same code is thus reusable over a wide range of types without *templates* (C++) or *interfaces* (Java). This enables Python's built-in *list* data structure to contain objects of any type, simply by enclosing them in square brackets, such as [" hello ", 3, 2.95].

Python is also object-oriented with its support for classes, inheritance, method invocation, and

[1] We mean Python 2.6 as of this writing, instead of Python 3.0, which is an *intentionally backwards incompatible* release.

instantiation. Objects may be instantiated simply by calling the constructor with the parameters, in the syntax of function calls.

Node Handles

A *node handle* is a data structure through which the program can access the node, analogous to a *file handle*. To create a node handle object in EcoCast, one instantiates the ecNode class with the statement var_name = ecNode(id), where var_name is the variable name that represents this node instance, and id is the node's network address. Subsequently, an application program can access the node by making Python method calls on the node handle object. Operations on attributes of such a node-handle object will have the same effect as directly manipulating them in the application program during runtime. One can use the object.method() and object.attribute syntax to invoke functions on the node and to access (get and set) attributes on the node. These accesses are translated into a sequence of actions on the host, ultimately reaching the nodes, and getting response back, as explained in Section 3.2.2. By treating nodes as objects, programmers can build complex applications such as graphical user interface or data analysis programs very easily without getting bogged down with the details of WSN programming.

For interactive mode, an expression is rendered to give the user instant feedback. For built-in types such as int and string, the values are rendered in the literal form such as 3 and "hello". For user-defined types (namely classes), this rendering can be accomplished by defining the special __repr__ method, which returns a string representation for the object's value. In our case, we render it as the constructor syntax with the value of the ID. For example,

```
>>> x = ecNode(123)    # instantiate a node handle
>>> x                  # prints the string returned by
ecNode(123)            # the x.__repr__() special method
>>>
```

Group Handles

In EcoCast, a *group handle* is used to construct a group of nodes. The ecGroup class is instantiated with the statement var_name = ecGroup(L) where L is a list of node IDs. The nodes in the group listen to a specific channel, and each node is assigned a group ID. The ecGroup is useful in functional programming constructs (Section 3.3.1), since a given group can be reused in several functional programming constructs without having to reconstruct a new group each time.

Functional Programming Constructs

Python provides **map**, **filter** , **reduce** as constructs for functional programming. EcoCast provides the corresponding functions for sensor nodes.

map(f, A) is equivalent to [f(A[0]), f(A[1]), f(A [2])...] , that is, forms a new list with the return values of calling function f on every element of the list A. A more general form of **map** takes multiple lists and is equivalent to calling f with arguments taken from the corresponding elements of the lists, e.g., [f(A[0], B[0], C[0]), f(A[1], B[1], C[1]), ...] .

filter (f, L) returns a new list of members x of L such that f(x) evaluates to true.

reduce(f, L) performs a binary operation f on the first two members of L, and applies the f operator on the result value with the subsequent member of L for the rest of the list. For example, if L has five elements, then it evaluates to f(f(f(f(L[0], L[1]), L[2]), L[3]), L[4]). If the optional third parameter Init is provided, then L[0] in the expanded expression is replaced with f(Init , L[0]) instead.

Functional programming constructs frequently make use of **lambda** expressions for defining anonymous functions for convenience. For instance, **lambda** x,y: x+y is an anonymous function that returns the sum of its two arguments. It can be passed in place of f to the **reduce** example above

to compute the sum of the list members.

We extend the same concept to operating on a list of nodes. Note that in the ecMap operator, f actually refers to a method rather than a function; both **filter** and **reduce** can be used instead of the mass version though much less efficient due to the lack of parallelism.

ecMap(f, NH) is equivalent to [NH[0].f(), NH[1].f(), ...] , that is, the list of return values from calling function f on every sensor node whose handle is in the list NH. A more general form of ecMap() takes additional lists whose members are passed as parameters to the calls.

ecFilter (f, NH) constructs a list from those elements NH[i] of NH for which f(NH[i]) returns true.

ecReduce(f, NH) calls function f on two arguments cumulatively on the node handle list NH from left to right, so as to reduce the list to a single value. An optional *initializer* can be passed to serve as an initial value.

In the these commands, the node handle list NH can be replaced by a group instance ecGroup to make it more efficient by reducing the time of automatically constructing a group corresponding to NH during the execution. We show a simple example using the commands above.

```
1 >>> listAll()
2 [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15]
3 >>> NH = ecGroup([1,2,4,5,7,9]) # creates group based on id
4 >>> NH
5 ecGroup([1, 2, 4, 5, 7, 9])
6 >>> ecMap(readTemp, NH)
7 [25.5, 25.3, 25.3, 25.5, 25.2, 25.3]
8 >>> ecFilter(lambda x: x.readAdc() < 2000, NH)
9 [ecNode(1), ecNode(2), ecNode(4), ecNode(9)]
```

Table 3.3: Scoping of symbols

Statement	Host scope	Node scope	Meaning
n = ecNode(**id**)	n, **id**		instantiate node handle whose address is **id**. call it n
x = n.readADC()	x	readADC	call readADC function on node n, assign return value to x

```
10 >>> ecReduce(lambda x, y: x + y.readTemp(), NH, 0)
11 177.5
```

In line 1, the function listAll () lists the all IDs of nodes which are found and bound to the host during the start up process of host system. Line 3 construct a group NH of nodes with ID [1,2,4,5,7,9] . If the user tries to access a node whose ID does not exist, an exception will be raised by EcoCast. Line 6 obtains readTemp function on each node in list NH to read the temperature and return a list of the results. Line 8 constructs a list of nodes from list NH whose result of readADC function is less than 2000. In line 10, the ecReduce command cumulatively adds the result of readTemp function from every node in list NH and gets the summary of temperature.

3.3.2 Esh Constructs

EcoCast provides a command line interpreter named Esh (for "EcoCast shell"). It accepts the full Python syntax in interactive mode, plus extended syntax for the purpose of mass programming (Section 3.5) and source-code inlining. Although users can directly load the class library and run everything from the Python prompt, the reason for this additional layer is to intercept constructs that must be processed before invoking the underlying Python interpreter. Specifically, inlined C code can be extracted and preprocessed this way. The rest of the classes can be used directly by any other Python program without going through Esh.

Job Management in Esh

Esh supports running background jobs while executing interactively. The followings are associated commands.

To run a job in the background, the user simply types the command as usual, followed by &, similar to most Unix-like shells. The jobs command can then be used to list the command names, execution status, and the result of all the background jobs. The getResult(**id**) command returns the execution result of a specific background job with *id*. For example:

```
>>> n.readTemp() &
>>> jobs

2 Background Job(s)
 No.            Command      Status    Result
=============================================
  1          n.readADC()       Done     2344
  2 ecMap(readTemp, nodes)    Running   None
>>>
```

Scoping commands

Esh provides commands that change the default scoping of the symbols so that those inside a designated node can be visible and accessed directly without having to qualify them with the node instance first. These can be very useful for executing a series of commands associated with a node or a list of nodes.

scope[*node instance* | *list of node instances* | ecGroup *instance*]

list of statements

end

When user enter the scope command followed by *a node instance* or *a list of node instances*, all the functions or variables in the commands entered later are mapped to the attribute resided on the sensor nodes.

While inside a scope block, The extern [*variable name*] command enables access to a *variable* in the global scope. Here is an example:

```
1  >>> t = 0      # global                  7  25.4
2  >>> n = ecNode(1)                       8  ... readADC() # n.readADC()
3  >>> scope n  # set scope to n           9  2433
4  ... extern t # access global            10 ... end
5  ... t =readTemp() # n.readTemp()        11 >>>
6  ... t
```

Line 1 initializes a global variable t. From line 5 to 10, all symbols except t refer to those inside node n, as specified by the scope statement on line 4 and extern t on line 5. Both readTemp() and readADC() on lines 6 and 9 refer to functions on node n and are equivalent to n.readTemp() and n.readADC(), respectively, without the scope command. Note that if n on line 4 is replaced with a list then each statement in the scope block turns into a **map** of the function to the elements (nodes) of the list.

3.3.3 Method Dispatching and Attribute Access

EcoCast performs three tasks as the language wrapper: type marshalling, code wrapping, and code swapping.

Table 3.4: Summary of EcoCast commands

Command	Type	Meaning
listAll ()	EcoCast API	list IDs of found nodes
listNei ()	EcoCast API	list all the node IDs and the neighbors of each node
broadcast ()	EcoCast API	find the network topology
path(s,d)	EcoCast API	return a route from the source to the destination
getNodes()	EcoCast API	obtain a list of node instances
ecNode	EcoCast class	open connection to one node
ecGraph()	EcoCast class	open the graph to the network
ecGroup()	EcoCast class	open group connections to nodes
ecMap()	EcoCast Functional programming API	**map** function for nodes
ecFilter ()	EcoCast Functional programming API	**filter** function for nodes
ecReduce()	EcoCast Functional programming API	**reduce** function for nodes
&	Esh qualifier	run a command as a background job
jobs	Esh command	show the information of background jobs
scope ... end	Esh block	set symbol scope to node or nodes
extern	Esh qualifier	enable access global variable in symbol scope of node or nodes

Type Marshalling

Marshalling is the task of ensuring that the sender or caller's data type is properly matched with the recipient or callee's, by conversion at runtime if necessary. EcoCast has access to all the prototype information for the functions and global variables, and it also can look up the architecture model for information such as the endian, word size, and their mapping to Python ones. Every native type of every architecture can be modeled as a Python class with the proper conversion operators. For instance, EcoCast can provide an unsigned, 16-bit, big-endian integer class as follows:

```
1 class UInt16big: # unsigned 16-bit, big-endian int
2     def __init__(self, n): # constructor
3         if (type(n) == type(1)): # type of 1 is int
4             # ensures value in range or throw exception
5             # store in self._rawData
6         elif # other types
```

```
7               # handle other types
8       def __int__(self): # typecast to built-in int
9               # form int from self._rawData and return
```

Note that by representing MCU data types this way in Python on the host, the underlying Python interpreter automatically invokes the proper type conversion operator accordingly. In the example of UInt16big, any value going to the node can be specified using just a generic **int** in Python; and any returning value automatically can be cast into the proper type, all without burdening the user with having to know all the type variants. For instance,

```
1 >>> n.setSamplesPerSec(20)
2 >>> n.bitResolution
3 12
4 >>> n.readADC()
5 15
```

The user does not need to be concerned with whether the samples per second parameter is represented as an 8-bit, 16-bit, or 32-bit signed or unsigned int, or even 12-bit or 14-bit words on some PIC MCUs. If the value is not one that can be handled by the underlying hardware, then the user gets an exception. Conversely, the user also does not need to worry about the data type returned by the readADC function – it automatically gets cast into the proper type as needed.

Code Wrapping

Code wrapping is a way for the node-handle object to abstract away implementation details as to *where* the code associated with a node object is implemented. Some code and attributes may be maintained (or "cached") on the host and therefore can be executed efficiently without incurring expensive wireless communication. In the example above, the setSamplesPerSec and readADC functions reside on the node, but both may be locally intercepted and optimized. This is because if there was a previous call to setSamplesPerSec with the same value, and no other host has altered the

samples per second attribute, then EcoCast can simply return without actually making the remote call. Similarly, if readADC is called five times within one second, but it had been configured to be sampling at once per minute, the user has the option of returning the cached value without communication. These are all analogous to how standard file I/O performs buffering on general-purpose computer systems.

Another form of abstraction is the support of convenient syntax for attribute access. If `n.bitResolution` is used as an R-value (i.e., on the right-hand-side of an assignment, or passed (by-value) as a parameter), then the underlying Python interpreter automatically calls the special method `n.__getattr__("bitResolution")` to compute the value. Similarly, if `n.bitResolution` is used as an L-value (i.e., in the context of `n.bitResolution = 12`), then Python calls the special method `n.__setattr__("bitResolution", 12)` automatically instead. This mechanism enables the node handle to appear as if it were the node itself.

3.4 Compilation and Linking

EcoCast performs compilation on two types of code for the node: C source code and Python code. C programs are written in a style that is native to the node's MCU architecture. Python code is normally interpreted in the context of the shell on the host PC itself, but the user can also write limited Python code to be executed on the node. EcoCast would rewrite the Python code into C and invoke the C compiler. Then, EcoCast processes the *map* file to perform linking incrementally so that a small patch file can be generated.

3.4.1 Compilation from C

C code can be written for low-level code such as device drivers, interrupt handlers, and accessing many hardware-specific features. C code can also be written for higher-level application features,

such as packet formatting, data filtering, etc. EcoCast does not actually compile the code; instead, it simply invokes whatever compiler is configured for this purpose. This way, the user has access to the entire C language and the library as supported by the chosen compiler.

In this implementation, we use the Keil compiler for 8051. The supported data types in C as of this writing include **char** (1 byte), **int** (2 bytes), **long**, **float** (4 bytes), and pointers (1, 2, and 4 bytes). For 8051, bit type is supported by hardware with the C (Carry) flag and bit-addressable registers, including GPIO ports. The EcoCast incremental linker generates directives to control the layout of the symbols, as explained in Section 3.4.3.

3.4.2 Compilation from Python

Instead of going to C language, the user can actually write restricted Python functions and expressions that are then translated into C to be compiled to run on the node. This is especially useful for specifying conditions such as threshold or triggering conditions, and these functions are often passed as the first parameter to a functional programming construct. We currently support named functions, anonymous functions (lambda), and bounded loops. A named function is defined as

```
def function_name(parameter_list):
        statements
        return expr
```

whereas a lambda expression represents an anonymous function as explained in Section 3.3.1. One problem is that Python parameters are dynamically typed and cannot be easily inferred. Our solution is to require the use of naming convention to encode the types of the parameters and the function. We put a prefix before parameter name to indicate the datatype:

Prefix	C_	UC_	I_	UI_	F_
C Type	**char**	**unsigned char**	**int**	**unsigned int**	**float**

The types of all local variables are automatically inferred and so as the return value. To facilitate type inference, we requires that the type of a variable to remain constant.

Arbitrary for-loops are not supported. However, simple for-loops over range are supported and are converted into efficient C counter-parts. Here is an example of Python-to-C translation.

Python	C

`for i in range(5):`
`  statements`
	`for (int i = 0; i < 5 ; i++) {`
	`  statements`
	`}`

3.4.3 Incremental Linking

Linking is the step of assigning addresses to symbols after the program source code has been compiled into object code. To minimize the size of the patching script, linking should be performed incrementally by considering the way the program has been linked in the previous version. For each node, EcoCast maintains a directory of the source files, memory map file, and command script of the node.

The map file contains information on memory usage, list of function segments, segment sizes and locations in program memory, and the symbol table. The function call tree is also assumed to be available with the proper compiler directives. Information obtained from the memory map file can only reveal the size and address of each function but not the actual function name, function parameters or return value. The information extracted from the map files are used by EcoCast for type marshalling and dispatching.

To minimize the size of the patching script, the incremental linker attempts to keep these assigned addresses unchanged between versions. The addresses include not only code but also data, constants, and library routines. If the linker can determine that two pieces of code are never invoked

together, then it may consider overlaying them when necessary to fit in the very limited amount of program memory while reducing the reference patching. Each code segment is given a priority based on the call tree generated by Keil C LX51 linker. The higher the priority, the smaller the number, starting from 1 (Priority 0 is assigned to driver code segments.) For each segment, the larger its call depth, the higher its priority. This ensures that segments that are least referenced be overlaid first. A least recently used (LRU) replacement algorithm is used to select among segments with the same priority.

3.4.4 Version Control and Patching Scripts

Conceptually, the host subsystem maintains a database entry for keeping track of the memory layout and source files of each sensor node. However, it is inefficient and not scalable to a large number of nodes, because we expect some or most nodes to have identical memory layouts. To eliminate redundant entries, we maintain the information according to the firmware version. Nodes with the same version share the same entry, and a new entry is created when functions are installed or updated, resulting in distinct memory layouts. This organization makes it efficient to re-compile, re-link, and updating code to a group of nodes at a time. Moreover, to encourage experimentation EcoCast integrates version control for the node firmware to enable roll back to any previous version of the firmware as simply as a simple "undo" command.

Upon generation of a new program image, binaries of the new function are extracted while pre-existing segments that are updated are *diff*ed with the original binaries to generate the patching binaries. These patches are then wrapped into store messages for installation or update on the target node.

3.5 Reprogramming and Execution

One user command on the host subsystem can be expanded into a sequence of actions on the host as described in the previous section. Then, they turn into messages that are received and executed by the nodes. The host first sends patching scripts to a node if necessary, and then sends the message to invoke the user's desired function. Moreover, EcoCast supports interaction with not only a single wireless sensor node but also an entire group of nodes. It attempts to parallelize their communication and execution as much as possible. This section first describes how the node handles messages, followed by group reprogramming over the air, and group invocation of functions with specific discussion on synchrony issues.

3.5.1 Execution Mechanism on the Node

The Node subsystem consists of two software components: *store* and *execute*. These two main components make use of the same underlying messaging mechanism for wireless communication.

Store Component

The store component is responsible for processing incoming data to the node. This happens during function installation or variable updates. A store message contains a 2-byte target address, 1-byte nonvolatile flag, a 1-byte length field, and the data payload. The nonvolatile flag indicates whether the data should be stored in nonvolatile memory such as EEPROM or flash, or if it should be written to RAM only. The difference is that the RAM content is lost when the device is rebooted. It is up to the programmer to decide whether the program is to be permanently updated or temporarily changed. In our case, **lambda** expressions are kept in RAM only.

Execute Handler

The execute component carries out programmer designated function, plus basic functions for memory access. It contains a set of *get* handlers that return memory content at a given address, and this mechanism can be further extended as a debugging utility. Unlike EcoExec, which assumes that the code must be copied from external EEPROM into on-chip RAM before it can execute, EcoCast assumes *execute-in-place* (XIP) model for program memory, although it is also possible execute from RAM for code that is meant to be run temporarily and discarded, such as most lambda expressions. In any case, the firmware is structured such that no need for rebooting. We achieve this by setting the highest priority on the code segment for the EcoCast runtime support such that it cannot be modified at run time.

3.5.2 Group Reprogramming of Nodes

To reprogram (patch) a set of nodes as opposed to an individual node, EcoCast performs the following steps, as shown in Fig. 3.3. In Step 1, the host individually sends JOIN message to ask the nodes that need to be reprogram to join the group. This group of nodes (the nodes with green spot shows in Fig. 3.3(b)) would listen to a specific channel and every member node keeps a bitmap to track packet loss. In Step 2, the host broadcasts the DATA messages via the specific channel on which only the group member can receive, as shown in Fig. 3.3(b). The DATA message includes the code data and the memory address to store the code. Each time a node receives a DATA message, it stores the code to the specific memory location and sets the corresponding bit of bitmap. After all DATA messages are sent, the host broadcasts a REQ message (Fig. 3.3(c)) in Step 3, and all the member nodes reply with a BMP message containing their bitmap to the host to indicate missing DATA messages. (Fig. 3.3(d)). A simple TDMA technique is applied to the BMPs from the nodes, where each node waits until its time slot as determined by its group ID and sends the BMP back to the host. The host rebroadcasts the lost DATA packets according to the BMPs and

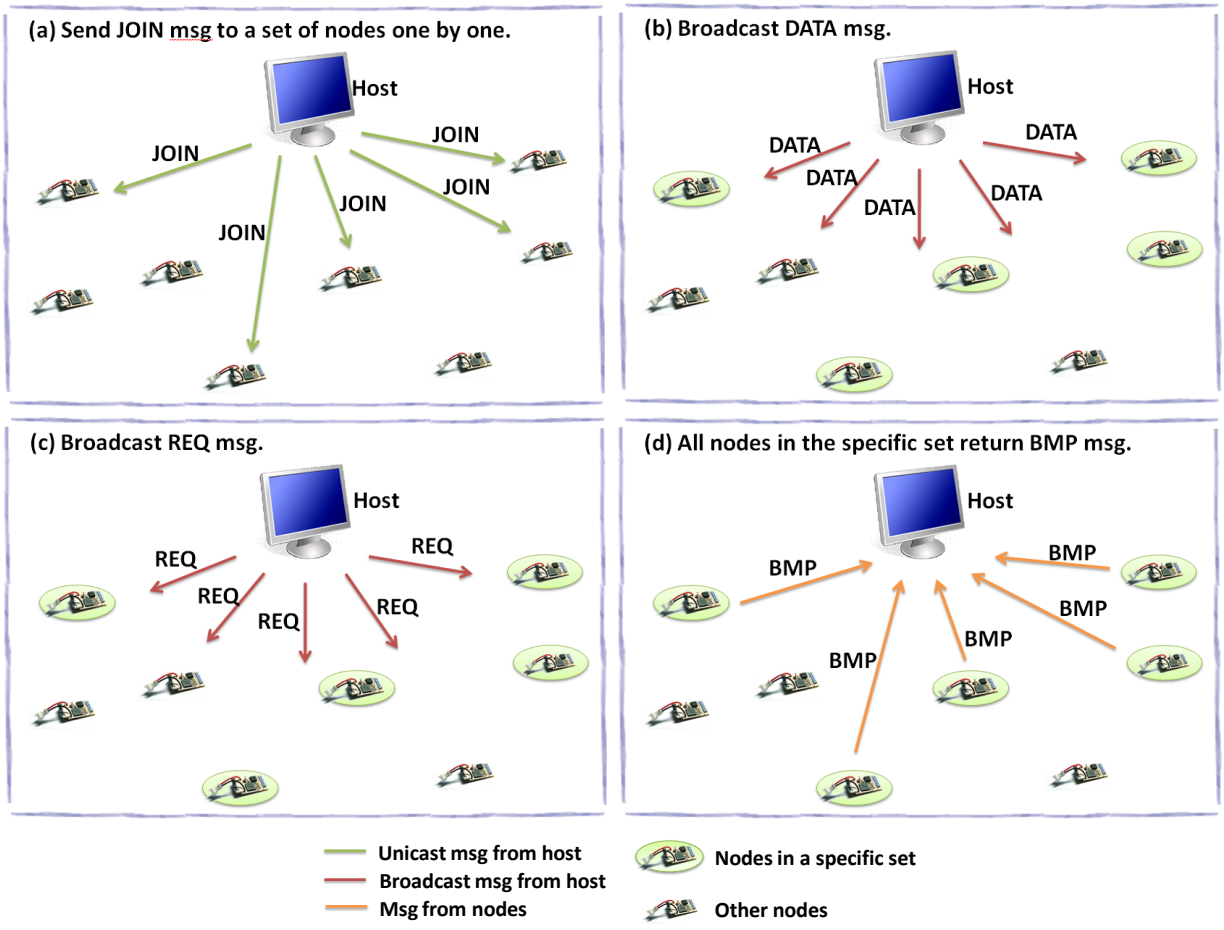


Figure 3.3: Reprogramming scenario

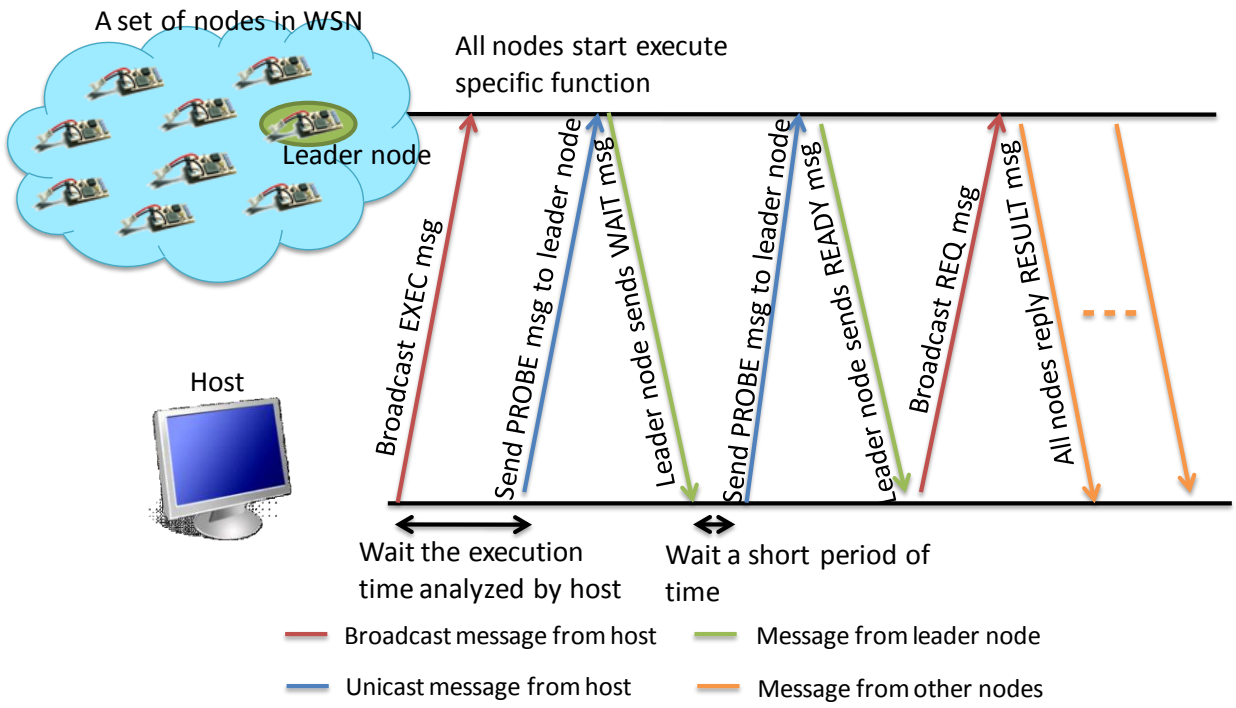


Figure 3.4: Function execution sketch

repeats the steps above until all members are reprogrammed correctly, unless the user decides to abort.

3.5.3 Group Execution of Functions on Nodes

To ask a node to execute a function, the host just needs to send an EXEC message to the node and wait until the node replies with the result. To ask multiple nodes to execute a function, such as the case with **map**, it would be inefficient to simply iterate over the nodes one at a time. Instead, the following protocol is used, as sketched in Fig. 3.4. Initially, the host broadcasts an EXEC message, and the nodes execute the specified function. However, at this point the nodes should not reply as soon as they are done, because it is likely to run into collision if not coordinated. The host is not blocked but may send a PROBE message to see if a node has completed the previous call. Instead of sending a PROBE to each node, the host sends it to only the node acting as the *leader*, since it is expected that all nodes would complete execution around the same time. The leader replies with

either a WAIT if it is still running and the host repeats; or the leader replies with a READY if it has finished. Then, the host broadcasts a REQ message and all the nodes return their execution results similar to the group reprogramming in Section 3.5.2. The nodes reply with RESULT message in a simple TDMA scheme, instead of the bitmap.

One question is how long the host should wait until it tries a PROBE. The overhead for the leader is high if we probe the leader too often, but if time interval between each probe is too long, the response latency increases. For this reason, a timing analysis tool named Bound-T [82] is used to give assistance. Bound-T computes an upper bound on the worst case execution time (WCET) of a program. EcoCast use Bound-T to analyze function execution time in order to perform a efficient probing scheme.

3.5.4 Background Job Management

To handle a background job, the interpreter assigns a thread to execute the command in the background, and the *background jobs manager* records the command, execution status and the information of the corresponding thread. When multiple jobs exist in the system, EcoCast is responsible for mapping the incoming packets from the nodes to the associated jobs on the host. Since every incoming packet is related to a node instance on the host system, the associated job is in charge of the packet. Note that we apply a locking scheme where an attribute of a node instance that resides on node can be access only by one job at a time to make the node preform the correct execution process. The jobs command is provided by EcoCast for the user to monitor the execution status and access the result after the background job is done.

3.6 Python Bluetooth Low Energy Wrapper

Base on what we have done in the EcoCast, python language does provide a more expressive way of development for these edge devices. There are still some improvements can be done in terms of better language binding. Also, when more and more edge devices are using BLE as their wireless communication protocol, it would be suitable to have python binding for the BLE.

Here we developed a new python base BLE peripheral wrapper called PyBLE. With the BLE data structure in mind, we have corresponded class design for the peripheral, service, and characteristics. A skeleton of profile can be described in Figure. 3.5. In a profile handler, the basic operations for a characteristic can be handled. Basic operations are

- Read
- Write
- Notification

The linkage of a profile hanlder and a profile on the BLE peripheral devices is done by assiging the UUID of the profile handler. Similar to the BLE structure design, profiles and characteristics are identified by UUIDs. This numerical identification scheme is not easy for people to comprehend, thus a naming scheme is also provided in PyBLE. When a peripheral is connected, we can use string to identified the profiles and enhence the readability of the source code as shown ing Figure. 3.6.

In EcoCast design, we basically describe the connected node into a programming language instance. we also bring in functional programming constructs to allow macroprogramming. In PyBLE, since the Python language has all features above and support natively, to have a fully Python compatible design is prefered. With a compatible python design, grouping, funciton programming constructs can be done in the scriting language level. Instead of mapping edge devices into in-

```
class ProfileHandler(object):
    __metaclass__ = ProfileHandlerMount
    names = {}

    def initialize(self):
        raise NotImplementedError

    def on_read(self, characteristic, data):
        pass

    def on_notify(self, characteristic, data):
        pass

    def on_write(self, characteristic, data):
        pass


class DefaultProfileHandler(ProfileHandler):
    UUID = "*"
    _AUTOLOAD = True

    def on_read(self, characteristic, data):
        ans = []
        for b in data:
            ans.append("0x%02X" % ord(b))
        return " ".join(ans)

    def on_notify(self, characteristic, data):
        print self.on_read(characteristic, data)
```

Figure 3.5: Profile handler skeleton and a default profile handler implementation.

```
def main():
    cm = pyble.CentralManager()
    if not cm.ready:
        return
    target = None
    while True:
        try:
            target = cm.startScan()
            if target and target.name == "EcoZe1":
                print target
                break
        except Exception as e:
            print e
    target.delegate = MyPeripheral
    p = cm.connectPeripheral(target)
    for service in p:
        print service
        for c in service:
            print c, " : ",
            print c.value

    c = p["EcoZen Profile"]["EcoZen Char 1"]
    print c.value
    c = p["FFA0"]["FFA1"]

    p["FFA0"]["FFA6"].notify = True
    c.value = bytearray(chr(1))

    cm.disconnectPeripheral(p)
```

Figure 3.6: PyBLE usage

```
class MyPeripheral(PeripheralHandler):

    def initialize(self):
        self.addProfileHandler(Acceleration)

    def on_connect(self):
        print self.peripheral, "connect"

    def on_disconnect(self):
        print self.peripheral, "disconnect"

    def on_rssi(self, value):
        print self.peripheral, " update RSSI:", value
```

Figure 3.7: A simple perpheral is described by PyBLE.

stances, the PyBLE is to describe a type of pheripheral as a class. When a peripheral is connected, user can map the connected pheripheral into a pheripheral class and instantiate the class. When disconnecting, the instance is destroyed. Then in the cyber world, the actual peripherals can be represented correctly. The peripheral class can also be reused. A simple implmentation of a actual peripheral is shown in Figure. 3.7.

Another advantage of using PyBLE is that composability of profiles and peripheral. The profile design is describe how a peripheral provides some data no matter what actual MEMs sensors are used in the system design level. So a peripheral can register multiple profiles to present what kinds of capabilities that this peripheral can provide. For example, a 3-axisal acceleration profile is impelmented in Figure. 3.8. All peripherals that provides this 3-axisal capability can be easily linked during the initialize phase. If an user wants to have a slightly different implmenetation such as different data processing, he can always inheret this acceleration profile and build their own handlers.

PyBLE comes with all SIG defined services and profiles in the library. Such as Heart Rate profile and battery profiles and ...etc.

```
class Acceleration(ProfileHandler):
    UUID = "FFA0"
    _AUTOLOAD = True
    names = {
        "FFA0": "3-axis Acceleration",
        "FFA1": "Sensor Enable",
        "FFA2": "Acceleration Rate",
        "FFA3": "X-axis",
        "FFA4": "Y-axis",
        "FFA5": "Z-axis",
        "FFA6": "All axis"
    }

    def initialize(self):
        pass

    def on_read(self, characteristic, data):
        ans = []
        for b in data:
            ans.append("%02X" % ord(b))
        ret = "0x" + "".join(ans)
        return ret

    def on_notify(self, characteristic, data):
        cUUID = characteristic.UUID
        if cUUID == "FFA6":
            x, y, z = self.handleXYZ(data)
            print x, y, z

    def handleXYZ(self, data):
        x, y, z = struct.unpack(">HHH", data)
        x = (0.0 + (x >> 4)) / 1000
        y = (0.0 + (y >> 4)) / 1000
        z = (0.0 + (z >> 4)) / 1000
        x =  2.0 if x >  2.0 else x
        x = -2.0 if x < -2.0 else x
        y =  2.0 if y >  2.0 else y
        y = -2.0 if y < -2.0 else y
        z =  2.0 if z >  2.0 else z
        z = -2.0 if z < -2.0 else z
        return (x, y, z)
```

Figure 3.8: An example implementation of a exisintg acceleration profile.

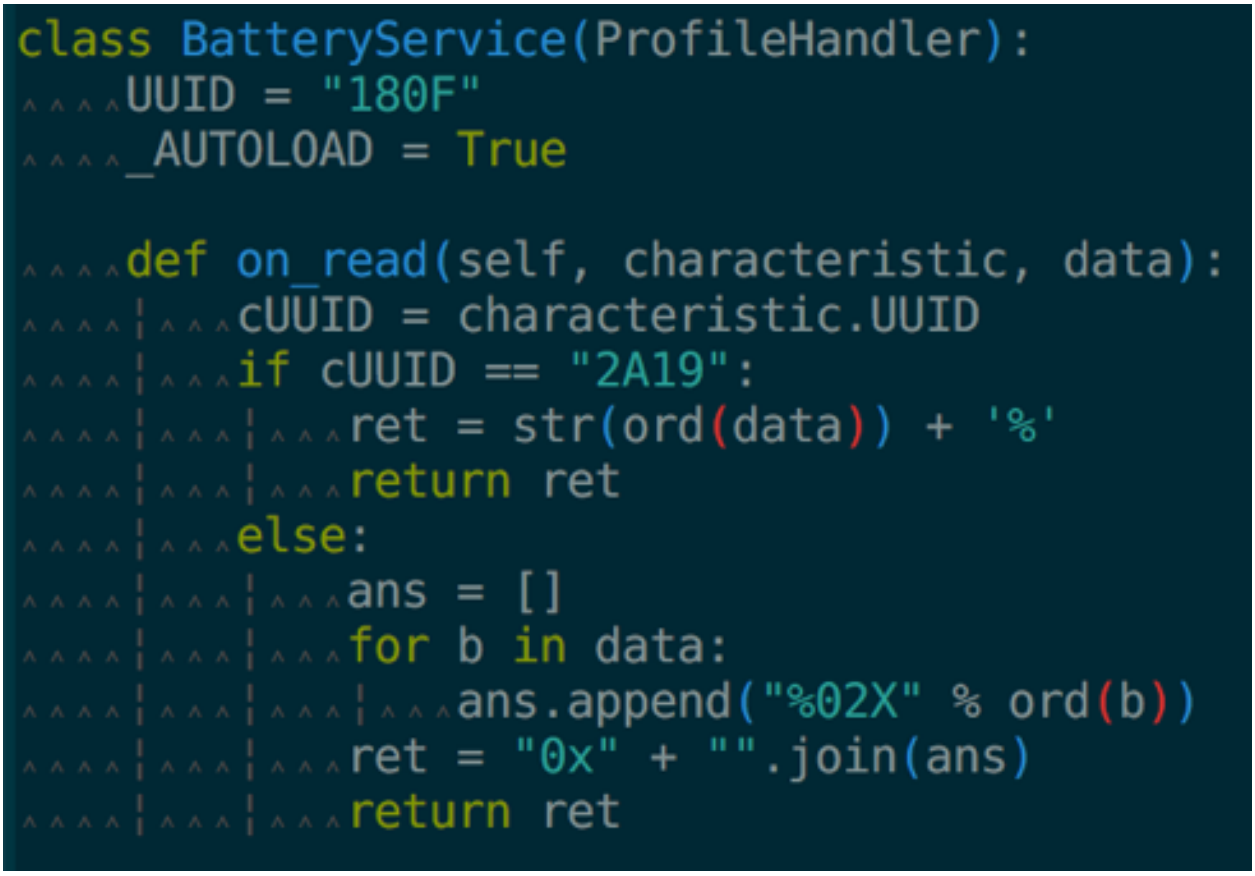


Figure 3.9: A BLE battery service hanlder implementation.

Table 3.5: Test cases of functional programming

Name	Description	Size[1]	Iterative command	Functional programming

TEMP	Read the output of temperature sensor and return to host.	202 bytes	`g = getNodes([0,1,2,3,4,5,6,7,8,9])`
			`for n in g:`
			`n.readTemp()`
				`g = ecGroup([0,1,2,3,4,5,6,7,8,9])`
				`ecMap(readTemp,g)`

ADC	Reads the output value for the digitalized output of triaxial accelerometer return to host.	212 bytes	`for n in g:` `n.readADC()`	ecMap(readADC,g)
ADC_1MIN	Samples the output value of triaxial accelerometer in 25Hz for one minute and return the average of the outputs.	372 bytes	No iterative version	ecMap(readADC1Min,g)
LIGHTING	Control the switch of the light(on/off)	12 bytes	n=Node(**id**) n.turnOn()/n.turnOff()	No Functional programming
DOORMON	Monitor the status of the door in eight seconds	417 bytes	n=Node(**id**) n.DoorMonitor()	No Functional programming

[1] The uploaded size is the size of patching binary generated by diff'ing the original binary with the new binary.

With PyBLE, a peripheral can be represented by some profile handlers. This allows users to replicate pheripherals in the gateway devices and reused on another gateway devices.

3.7 Evaluation

3.7.1 Experimental Setup

Our setup consists of a PC acting as the host connected through an Ethernet base station to ten nodes as described in Section 3.2.1. The nodes are all equipped with an on-board triaxial accelerometer, and different nodes have their output pins connected to actuators for lighting control. At the beginning, all nodes have been programmed with only the essential drivers, including SPI, flash memory, and RF, but no application code. Each application is incremental linked and uploaded by EcoCast on demand.

3.7.2 Functional Programming

Three applications are created and the details are shown in Table 3.5. All the nodes are in the single-hop range and the RF transmission power is 0dBm. We use these applications to evaluate the performance of parallelizing remote reprogramming and function execution of EcoCast. In this section, we evaluate the performance of functional programming constructs of EcoCast in terms of

reprogramming latency and detailed breakdown of round-trip command latency.

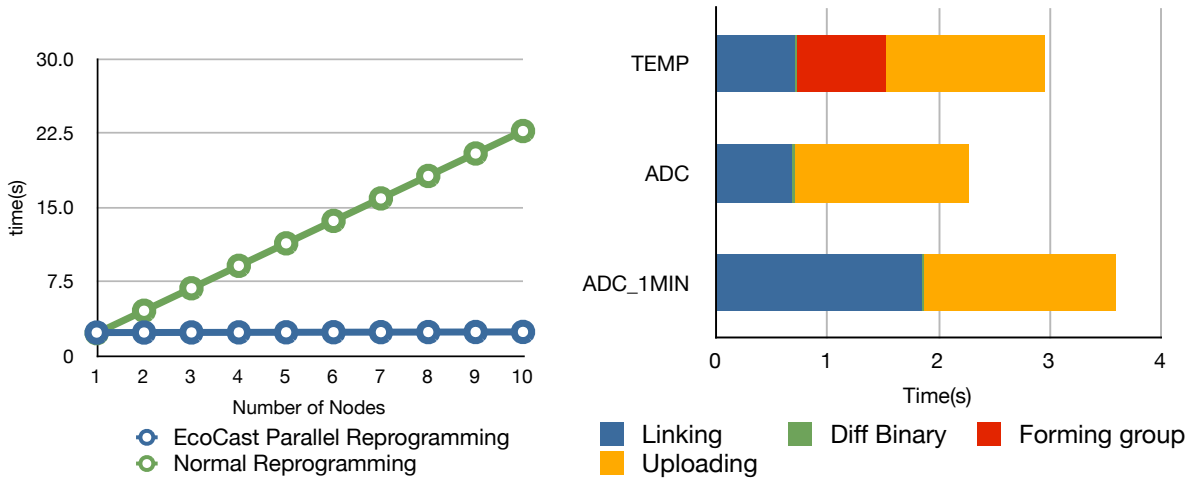
Reprogramming Latency

EcoCast performs group reprogramming to reduce the reprogramming latency while reprogramming multiple nodes. Fig. 3.10(a) shows the comparison of latency between sequential reprogramming and group reprogramming by EcoCast for installing the ADC application on-the-fly. Installation of a new application entails several steps, including linking, diff binary, uploading, and group forming if reprogramming a group of nodes. We demonstrate the reprogramming latency of applications TEMP, ADC and ADC_1MIN by group reprogramming of EcoCast where the size of group is 10 nodes. The applications are uploaded in sequence. Fig. 3.10(b) shows the latency of different steps. Note that ADC and ADC_1MIN do not have time of forming group because they can reuse the same group. The linking time of ADC_1MIN is longer since EcoCast tries to keeping the shared functions of the two ADC applications at the same locations. On average, it takes about 6.15 ms to upload on byte and write it into flash memory. In our experiments, the compilation-linking-installation-confirmation procedure require less than 4 seconds total for 10 nodes, which is acceptable. Several other wireless reprogramming schemes actually do not confirm if the nodes have been programmed successfully.

Round-Trip Command Execution Latency

The round-trip command execution latency, also called the response time, is defined from issuing the command to receiving the results back on the host. We measure the response time of TEMP and ADC by serial execution iteratively (i.e., for-loop) and in parallel (using ecMap) over a range of group sizes from 1 to 10. We execute each application 50 times and record the average response time. The result is shown in Fig. 3.12.

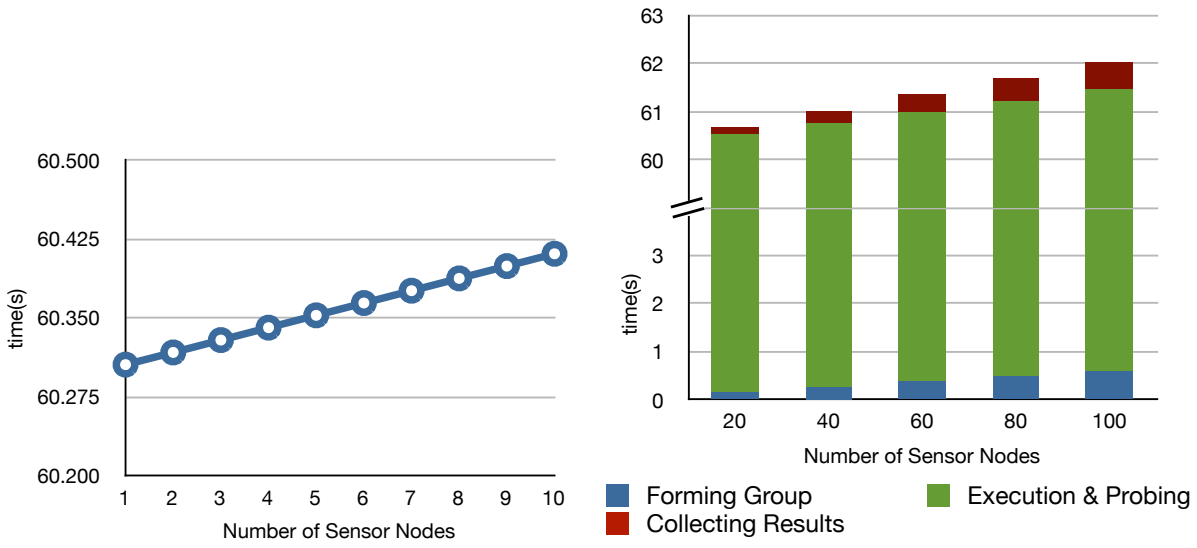
As measured in experiment, the runtime scales linearly for serial execution. For parallel execution,



(a) Reprogramming time of ADC.

(b) Ratio of different steps of latency in reprogramming 10 nodes

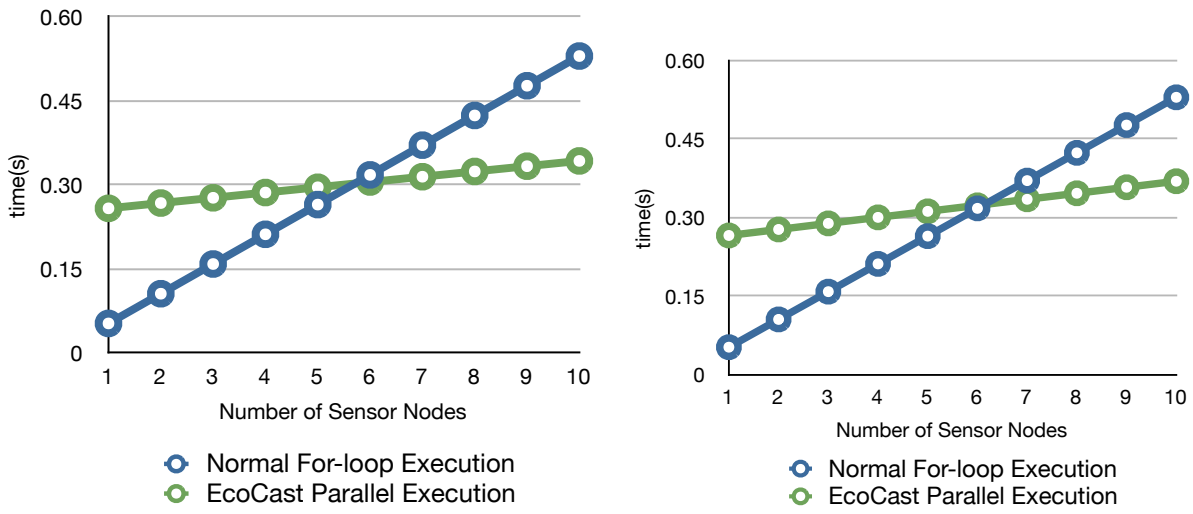
Figure 3.10: Reprogramming Latency



(a) Response time

(b) Ratio of different phases

Figure 3.11: Response time of ADC parallel execution of EcoCast



(a) TEMP (b) ADC

Figure 3.12: Response time of serial execution and parallel execution of EcoCast

we plot the worst case that includes the additional phase of group forming before execution, even though for the common case the same group can be reused. With this assumption, The response time of serial execution exceeds that of parallel execution for group sizes of 6 and higher in both TEMP and ADC applications. This means the group overhead can be amortized even for a modest number of nodes even for very short execution delays. Fig. 3.13 shows the breakdown of parallel (ecMap) execution times of ADC into three phases: *forming group*, *execution and probing*, and *collecting results*. The group formation time can be eliminated by reusing the same group, while the *execution and probing* phase remain nearly constant. The *collecting results* phase would still grow linearly but at a much lower slope. For applications with long execution times, the times of *forming group* and *collecting results* phases become negligible. The measured result of ADC_1MIN shows in Fig. 3.11(a). By following the TDMA schedule, we estimate the response time with large group sizes according to the trend of Fig. 3.11(a) and show the time of different phases in Fig. 3.11(b). We observe that *forming group* phase and *collecting results* phase take only 1.18s in a group with 100 nodes and represents a small portion of the total response time.

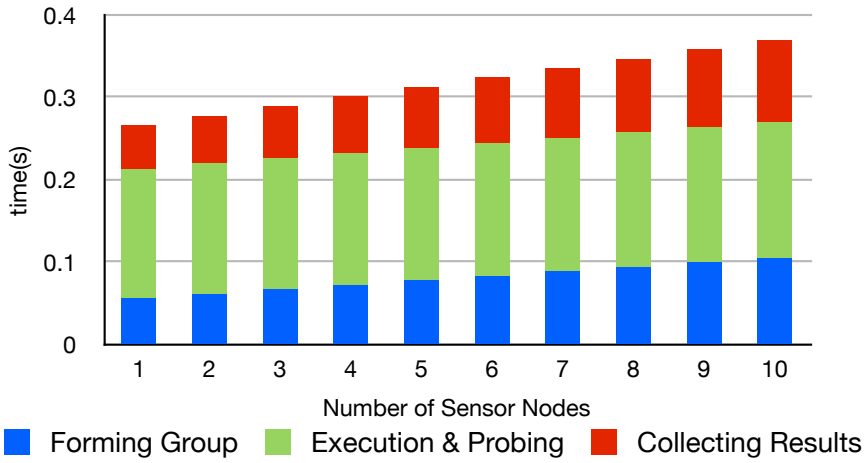


Figure 3.13: Ratio of different phases to execution time of ADC via parallel execution of EcoCast.

3.7.3 Multi-hop Networking

In this section, we show the evaluation results of our multi-hop networking method. Topology discovery was done by calling *broadcast()* on nodes in breadth-first order. A distributed scheme is possible, but we perform a centralized BFS on the host as a test case along with lighting control and door monitoring to demonstrate interactive command issuing over the multi-hop network. LIGHTING is for lighting control and supports commands for turn on/off a light. DOORMON is for door monitoring by checking if the door is open or closed. We evaluate their performance in terms of the latency.

Network Establishment

To stress test topology discovery, we spread out the ten nodes about 30 cm apart while setting the RF transmission to the minimum level of -20 dBm. The rather low signal-to-noise ratio (SNR) result in more bit errors in the RF transmission. We test many rounds when finding the topology information, and Table 3.6 shows six rounds. The union of the connectivity graphs is shown in Fig. 3.14. Due to packet loss, it is necessary to continue retrying finding neighbors a number of times to wait for ACKs before giving up, and we set the timeout to 1 second. It takes 9.5 to

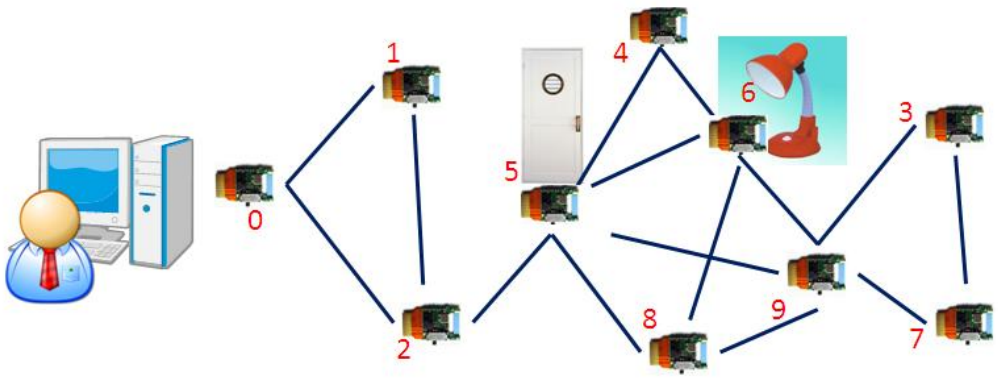


Figure 3.14: Union of connectivity graphs

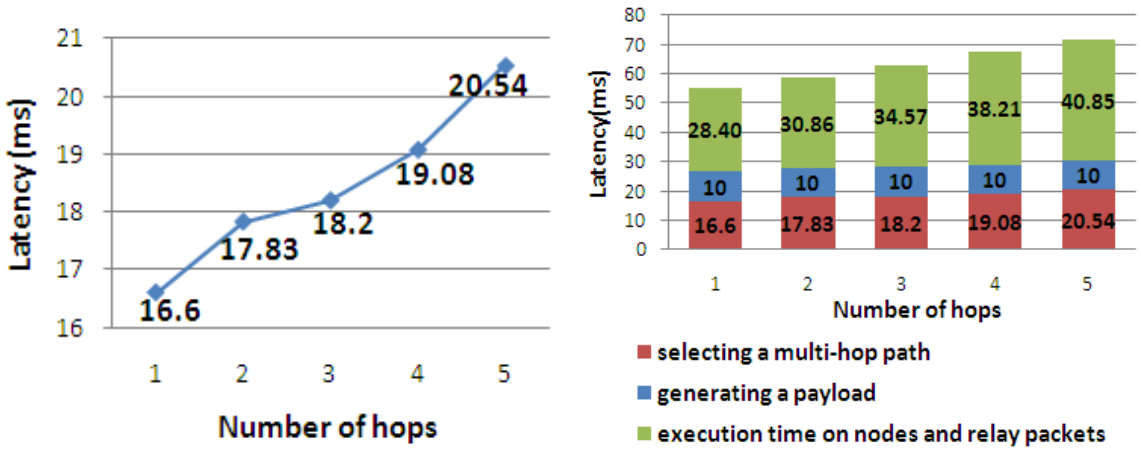
20.7 seconds, depending on the specific RF condition. With higher SNR and shorter timeout, the topology discovery time can be reduced by an order of magnitude. The maximum number of hops in all discovered topologies is five.

Reprogramming Latency

The nodes did not have the application code for LIGHTING or DOORMON in the firmware before deployment. Upon invoking the turnOn() and turnOff () functions for LIGHTING, EcoCast prepares the binary for the functions, each of which is six bytes, and transmits two packets to the target node, which patches its own firmware and executes. As shown in Fig. 3.16(a), it takes 35.9 ms to relay a packet and reprogram the node that is one hop away from the host, with an average of 2 ms for each additional hop. For the DOORMON application, the code size is 417 bytes, which must be divided into 30 packets to the nodes and must be fully acknowledged. As Fig. 3.16(b) shows, it takes 1.99 seconds to reprogram a node that is one hop away and 2.31 seconds for five hops away.

Round-Trip Execution Latency

The total response time is the sum of delays for selecting a multi-hop path, generating the payload, relaying a packet, executing the handlers, and transmitting the reply packet back, as shown in Fig.



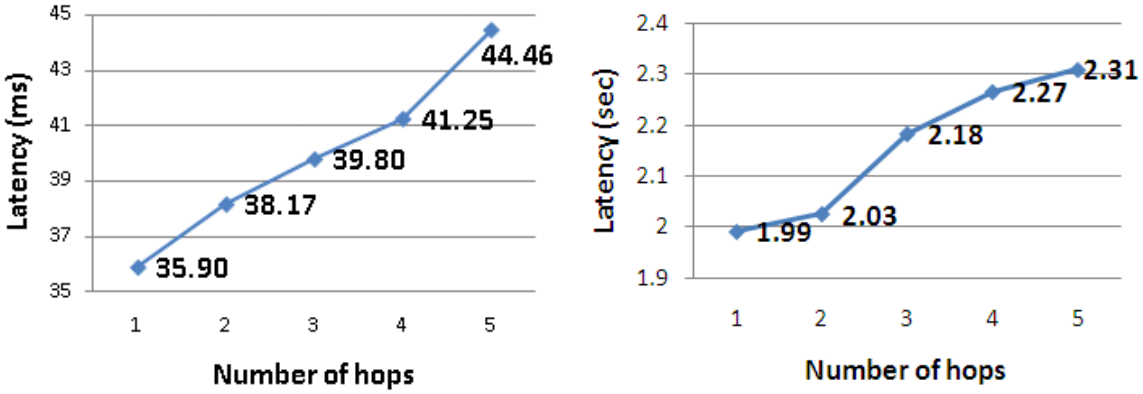
(a) (b)

Figure 3.15: (a) Latency of finding a path for each node, (b) Response time of calling LIGHTING function.

Table 3.6: The Latency of topology discovery phase

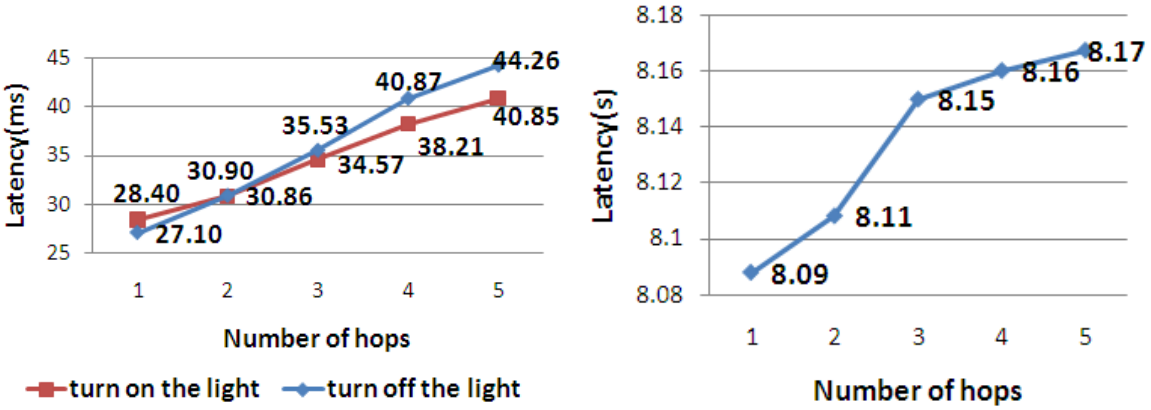
Round	Latency(sec)	Neighbors of each node
1	20.789	0: [1, 2], 1: [0, 2], 2: [0, 1, 5], 3: [], 4: [], 5: [2, 4, 6, 8, 9], 6: [4, 5, 8, 9], 7: [3, 9], 8: [5, 6, 9], 9: [3, 5, 6, 7, 8]
2	9.534	0: [1, 2], 1: [0, 2], 2: [0, 5], 3: [], 4: [1, 5], 5: [2, 4, 6, 8, 9], 6: [4, 5, 8], 7: [3, 9], 8: [5, 6], 9: [5, 7]
3	17.694	0: [1, 2], 1: [0, 2, 5], 2: [0], 3: [], 4: [5, 7, 8, 9], 5: [1, 4, 6, 8, 9], 6: [5, 9], 7: [3, 4, 5], 8: [], 9: [4, 6, 8]
4	17.713	0: [1, 2, 5], 1: [0, 2, 5], 2: [0, 1, 5], 3: [], 4: [3, 5, 7, 8], 5: [0, 2, 4, 6, 8, 9], 6: [5, 8, 9], 7: [3, 4], 8: [4, 5, 6, 9], 9: []
5	15.537	0: [1, 2], 1: [0, 2, 5], 2: [0, 1, 5], 4: [], 5: [1, 2, 4, 6, 8, 9], 6: [4, 5, 8, 9], 8: [4, 5, 6, 9], 9: []
6	12.368	0: [2], 1: [0, 2, 5], 2: [0, 1, 5], 4: [5, 6, 7, 8], 5: [1, 2, 4, 6, 8, 9], 6: [4, 5, 8, 9], 7: [4, 5], 8: [4, 5, 6, 9], 9: [5, 8]

3.15(b). To invoke the turnOn() function in the LIGHTING example, it takes 16.6 ms to find a path, 10 ms to generate the payload, and 28.40 ms to relay the packet and execute the handler on the node. That is, the response time is dominated by the transmission and execution latency, especially as the hop-count increases. While it is possible to continue increasing the hop count, the user may want to consider the practicality aspect and whether the added latency is acceptable when lowering the transmission power level.



(a) Light control (b) Door monitoring

Figure 3.16: The latency of adding functions



(a) Light control (b) Door monitoring

Figure 3.17: The latency of calling functions

Table 3.7: Comparison of Memory footprint (bytes)

OS or Runtime	Program memory	Data memory
Maté[53]	39746	3196
SOS core[27]	20464	1163
TinyOS with Deluge[33]	21132	597
LiteOS[10]	30822	1633
Mantis OS[**?**]	$\approx$ 14000	$\leq$ 500
EcoCast	$\approx$ 5200	$\approx$ 300

3.7.4 Memory Footprint on the Node

EcoCast can be configured with or without multi-hop and drivers. The core occupies 1550 bytes of program memory and 200 bytes of data memory. With the drivers, EcoCast takes about 4000 bytes of program memory and 250 bytes of data memory, still significantly smaller than other works. The multi-hop takes 1200 bytes of program memory and 58 bytes of data memory. Table 3.7 shows a comparison of memory footprint.

3.7.5 Discussion

The experimental results show parallel execution of functional programming constructs to work well on single-hop network; the performance is adequate for a modest-sized multi-hop network, but there is plenty of room for optimizations. One way to speed up is pipelined execution, although this is much more difficult in practice than in theory due to wireless communication. Another way is to explore asymmetric routes. If some nodes are not part of the targeted group for function invocation, then perhaps they can play more of a relay role, and reply packets need not take the reverse routes as request ones. The more even distribution of workload may also have energy efficiency benefits. Many more optimizations are possible for filter and reduce constructs in a multi-hop network. The contribution of this work is not so much the relatively simple multi-hop protocol but the interactive execution framework, on which we hope many innovative ideas at these higher levels can be implemented quickly and put into practice.

Chapter 4

A Modular Backend Computing System for Continuous Civil Structural Health Monitoring

4.1 Background

4.1.1 Event Triggered v.s. Continous Monitoring

Sensor systems for structural health monitoring (SHM) can be categorized into *event-triggered* and *continuous monitoring* systems. An event-triggered one wakes up sensors to collect data in the presence of some pre-defined events, such as earthquakes and heavy winds. In contrast, a continuous monitoring system collects all data samples. In general, SHM systems require sampling rates on the order of 1000 Hz and can be considered continuous monitoring systems. However, Depending on the response time and battery-life requirement, the system may perform undersampling, i.e., by sampling at the full 1000 Hz rate only for a few minutes per day and sampling at

50 Hz or lower at other times. One such system is Pakzad[66] which has deployed a total of 64 nodes on the Golden Gate Bridge in San Francisco, California to test the scalability and performance of the wireless sensor network. Each node is based on the commercially available MicaZ platform with two accelerometers and one temperature sensor. Ho[30] deployed a solar-powered vibration and impedance sensor based on the iMote2 on the cable-stayed Hwamyeong Bridge in Busan, South Korea and evaluated hybrid SHM capability, solar-powered operation, and wireless communication.

4.1.2 PipeTECT system for SHM and Water Pipe Monitoring

In our previous papers[79][45][44], we have developed a continuous monitoring system called PipeTECT to monitor disaster events on water pipelines. The smart sensors, called DuraMotes, perform noninvasive monitoring on the exterior of the water pipes by measuring its vibration at 1000 Hz. The same sensing mechanism can be applied to civil structure health monitoring as well, such as buildings and bridges. The PipeTECT system has a backend computing system for controlling sensing system, storing acquired data, data post-processing, and data dissemination. We have design the backend system to provide a web service for users. This makes the system accessible on regular personal computers and also modern mobile devices with Internet connectivity. As with many experimental systems, the massive amount of data generated by our sensing systems on pipelines and other civil structures were archived on our backend server as files. To discover a sudden disaster event in time domain, continuous monitoring was necessary, since events could not be readily detected by the sensors themselves without a global analysis of data from different sensors in the network. Due to the limited number of accessible locations (e.g., manholes or fire hydrants) for noninvasive water pipe monitoring, the sensors cannot be densely deployed but are often kilometers apart, one of the fundamental tasks is rupture localization, which is to pinpoint where rupture occurs. Unlike triangulation, which can be done as collaborative event detection, the backend server collects vibration data from sensors deployed over the entire city and computes

the pressure change gradient on a hydraulic model. Subsequently, the system must trigger a notification to administrator for how to respond to the event if rules for automatic response have not been defined.

One issue with the backend is how data are stored and handled. The web server is the universal way for the dissemi- nation of all types of data, including sensor data, and the natural way is to organize the data as files whose names encode the date, time, location, and other metadata. The data may even be in human-readable text for manual inspection. Also, if plain text files are used instead of self-descriptive formats such as XML, then the processing algorithm may need to load in different parsers specific to each fileâĂŹs data format or risk incorrect data interpretation. However, it can be inefficient to search the very large volume of data, and the text representation can also be space requirement. Although text files are easy to compress, it will only exacerbate the search problem.

4.2 System

Our PipeTECT monitoring system can be divided into *sensing* and *backend* subsystems. The nodes in our sensing subsystem form a tiered network, where the sensing tier is wired for underground operation, and the aggregation tier is wireless for above-ground operation. Our backend system is further divided into three subsystems base on functionality: *mudular kernel*, *data collection* and *storage*, and data dissemination. We briefly review our sensing system first and then describe our backend system.

4.2.1 DuraMote Smart Sensor

DuraMote is the name of our tiered networked sensing system. The two tiers are the sensing tier and the aggregation tier. Because the system was originally designed for water pipeline monitoring, and most water pipes are underground, the sensing tier must be able to transmit data and power in

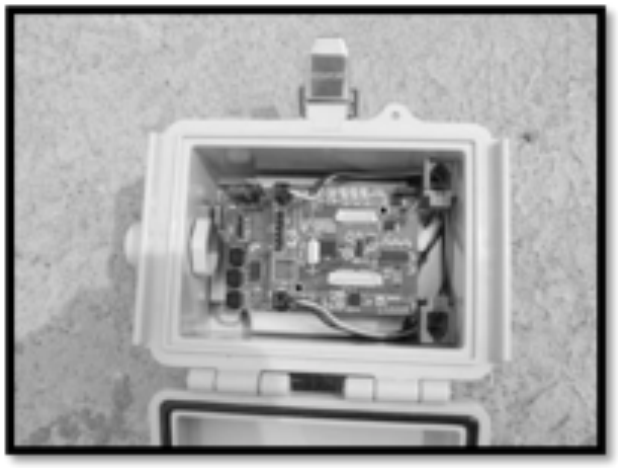


Figure 4.1: Gopher

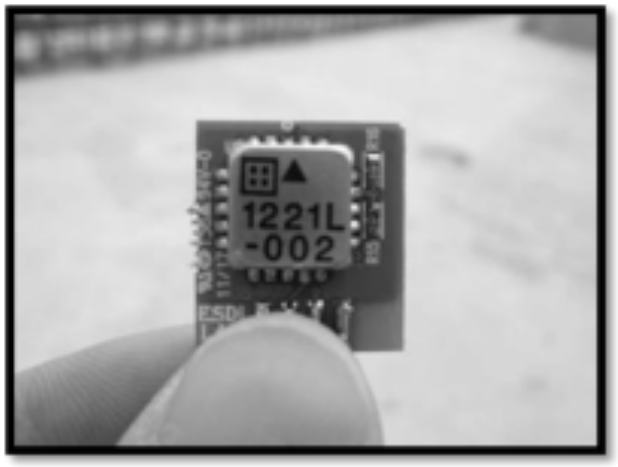


Figure 4.2: Accelerometer

underground settings. Therefore, a wired network solution is used for the sensing tier. Multiple sensors in a sensing tier can be connected to a data aggregator node at the sensing tier, whose nodes can collaborate to route data to or from an Internet uplink. Because the data aggregators are far apart from each other, a wireless network is more practical in terms of ease and cost of deployment. To support this tiered network operation, we design two types of nodes: a sensing node named *Gopher* and an aggregator node named *Roocas*.

A Gopher node as shown Figure. 4.1 is the wired smart sensor to be deployed directly on the exterior of pipelines or on civil structures for non-invasive monitoring. It contains up to three uniaxial

Figure 4.3: Roocas

MEMS accelerometers (SD1221L-002) as shown in Figure. 4.2, where the Z-axis accelerometer is soldered on-board while the X and Y axes ones are of removable type. The main board also includes a tilt sensor and expansion sockets for connecting other sensors, such as humidity, pressure, and gas. The sensor signals are digitized by the QuickFilter chip (QF4A512), which contains a 4-channel, 16-bit analog-to-digital converter with digital signal conditioning functions. It can be configured as a band-pass filter for different applications and generate different sampling rate accordingly. The Gopher nodes support Controller Area Network (CAN) as its primary data communication interface in daisy-chain topology in the sensing tier. One end of the daisy chain is a data aggregator for Internet uplink. The sensor nodes transmit sensing data via CAN bus upstream through the data aggregator, which has the option of logging data in its own memory card and processing before forwarding the data to the backend. Data can also travel downstream for commands and system administration.

A Roocas node as shown in Figure. 4.3 is a data aggregator node without sensors. It contains communication interfaces for downstream, upstream, and in-tier communication. The primary downstream interface to Gophers is CAN, which is used for both data communication and power. The primary in-tier and upstream interface is Wi-Fi (802.11n, up to 250 m) by default, although Ethernet is also an option. In addition, the Roocas board contains expansion interfaces for other

wireless modules, including XBee (up to 1 km range), XStream (by Digi, up to 64 km range), and Bluetooth Low Energy (BLE). The XBee and XStream can serve as the protocol for the aggregation tier, whereas BLE is more suitable for downstream, but there are no inherent limitations. Another common use of the Roocas is data logging onto a Secure Digital (SD) flash memory card when an uplink to the backend is unavailable. We use the RJ-9 modular connector commonly used for telephones as the wires for our CAN bus. As RJ-9 cables contain four conductors in a bundle, we use two of the wires for data and two for power distribution. Each Roocas node can connect up to 4 Gopher nodes in daisy-chain topology due to bandwidth limits of 1 KHz sampling rate, even though CAN itself can support a much larger network. The length of cable is up to 25 m between Roocas and Gopher, mainly due to cable resistance. If a longer length is desired, then cables of better quality or an extra cable must be used to reduce the voltage drop. This design enables a Gopher node to reach an underground pipeline covered by a metal lid or sealed confined spaces while the Roocas is deployed aboveground with a steady power supply and Internet uplink.

4.2.2 Backend System

We started out our backend system design with following goals in mind: cross-platform, universal accessibility, dynamic software module loading, fast data storage service, visual representation, and service mashup capability. Our backend system can be viewed as a next-generation supervisory control and data acquisition (SCADA) system, but instead of being implemented in native code only for one operating system, our backend system is executable on Windows, Linux, and Mac operating systems. We achieve the cross-platform interoperability by implementing most backend features with the Python programming language, which runs on most operating systems. On the front end, we build our user interface as a web application to be access from multiple devices with minimum effort. We now describe the subsystems in the backend, including the kernel, data collection and storage, and data dissemination.

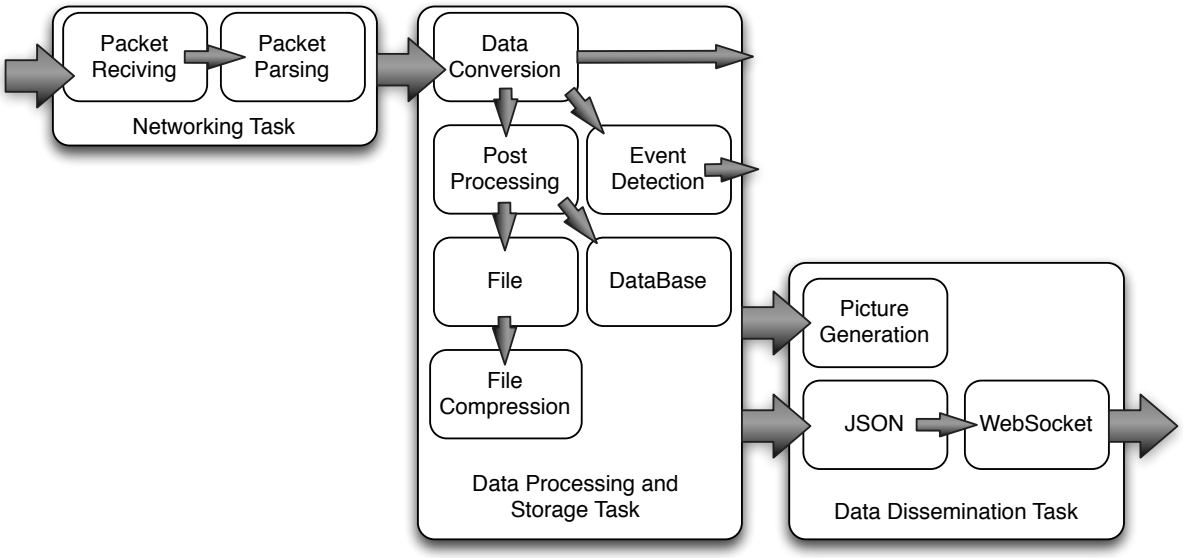


Figure 4.4: Overview of Our Backend System

Kernel

The kernel subsystem is responsible for coordinating the flow of data in our backend system by dispatching the associated tasks in each stage, as illustrated in Figure. 4.4. The basic tasks are networking, data collection and storage, and data dissemination, while additional tasks can also be loaded by the dispatcher as needed. The kernel dispatches the networking task to receive data packets from the sensing subsystem. Then, the kernel dispatches the data collection and storage task to process the received data and store the results. Once the data or results are available, the data dissemination task renders them in different forms as requested.

Each task is a standalone process and communicates with another task through message queues. Unlike a process (in operating systems) that needs to be spawned and is destroyed upon completion, tasks (in our kernel) always exist in the system and are awaiting messages. In a task, there are one or more software components. These components are loaded when a new message received by the task. The execution sequence of these software components is either sequential or parallel base on the sequence of execution. Sequential software components can be grouped as a thread.

When a data is placed on the message queue, the dispatcher will check if the recipient task exists in the system and loads those that will be needed. Upon receiving a message, the task process loads all registered software components on the fly. This dynamic loading into the plugin system of the kernel is accomplished by Python's metaclass[81]. With the plugin system, new tasks and software components will be loaded dynamically without requiring system reboot. The plugin system also checks if updated tasks exist by checking the object hash of the task and components. While software components are loaded when needed, It takes more time to fully replace a task on the fly and may not be fast enough for a continuous monitoring system. Thus, the updated tasks would be invoked first and start handling incoming data, and the dispatcher would first disconnect the message queue to the old task. The old task would continue to exist after current data is handled.

Data Collection and Storage

The data collection and storage subsystem is divided into two tasks: network communication and data processing. The former interfaces with the sensor network, while the latter handles data storage as well as applying algorithms on the data.

The kernel dispatches the networking task when a data packet is received on the backend systemâĂŹs network interface from the PipeTECT sensing subsystem. The data packet is first cached in the message queue before a computation unit becomes available in the networking task. Then, the networking task parses the packet into multiple data records according to the pre-defined format and puts them in the outgoing message queue to the next task, so that they are ready for further processing. There may be multiple network interfaces with different message formats and communication protocols, but usually only one networking task exists in the backend system.

The data-processing task inputs a set of received data records and applies processing algorithms to extract higher-level knowledge about the structure being monitored such as its structural health, remaining service lifetime, residual payload capacity, or the rupture location. The task can also

store the raw or processed data on the backend computer's disk space or send them to the dissemination task, which can render the data in different forms to users on other computing systems. The data processing task first converts the data records into the appropriate unit before sending data records to other tasks or other software components in the task. The converted data may be sent directly to the data dissemination task for visual rendering, to the file logging component, or to the event-detection component to see if a warning message need to be issued to administrators. Data processing in a task can be executed in parallel depending on the knowledge distilled from the collected data.

Data Dissemination

The data dissemination subsystem is responsible for not only making raw and computed data available for download but also for handling queries into the data sets and rendering them for visual presentation. For a continuous monitoring system, the raw data should be archived and made available to researchers and future investigators, and in the most extreme cases, the data should be made available in entirety. However, most of the time, the user may be interested in only selected segments that satisfy the filtering criteria; in fact, even with filtering, the number of data records may still be too large to show in the raw form. However, limiting the data to a small subset may be insufficient for detecting long-term trends. Today, most such visualization systems are implemented today as a standalone window-system application. In contrast, we start our dissemination system as a web service, which is readily accessible by all systems with a browser.

The first generation of our graphical user interface is based on HTML and static image files generation. Generated files can be put on a existing web server and be ready for viewing by users. Figure. 4.5 show example screenshots of static PNG images generated by the data dissemination process using the matplotlib[34] Python library to visualize the sensing data collected by Gopher nodes. The generation time of an image depends on the size of the collected data and resolution of the result image. However, a web-based image presentation may not be suitable for mobile devices

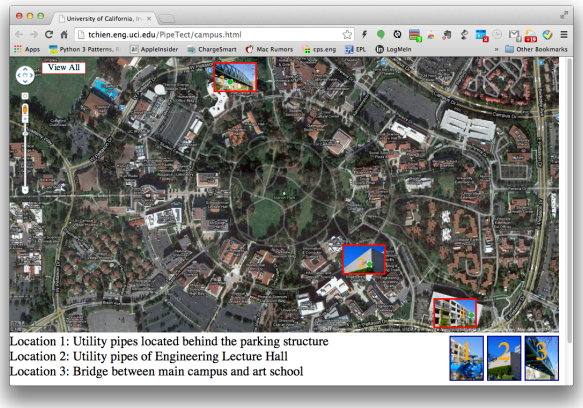


Figure 4.5: Campus

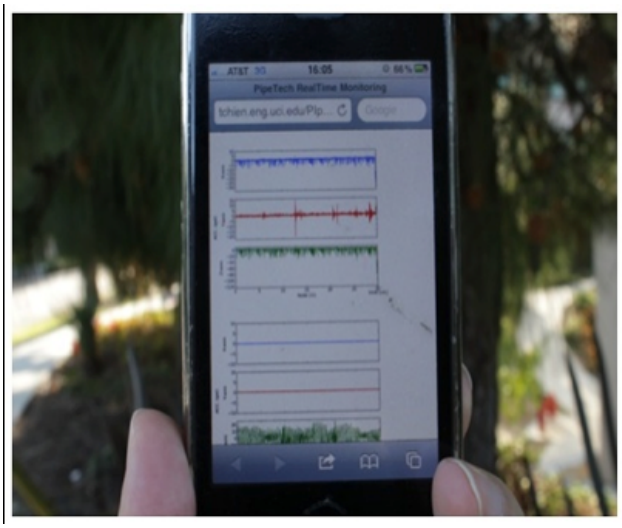


Figure 4.6: Using Smartphone to show the monitoring system

due to the limited screen size. For a smoother user experience, the visual presentation needs to be manually adjusted based on the web page's refresh rate, the amount of sensing data, and the resolution of the generated image, but this manual tuning process is time-consuming. An acceptable configuration is to have a 5-second refresh time interval, 30 seconds of sensing data, and an 800 X 600 image resolution for a 1024 X 768 web page. User could also view the content on a mobile device such as smart phones or tablet, but the configuration is not optimized for this kind of usage in Figure. 4.6. In any case, the first generation suffers from a rough user experience, because images fetched from a browser cannot be perfectly synchronized with frequent image generation.

The second generation of the system improves the user experience by means of dynamically generated web contents instead of static files. That is, the backend system serves as a content provider rather than a static file generator, and much of the data rendering is done locally by the user's browser. We use the Python Tornado Web Server and enable picture generation by the user's browser using a Javascript library named Flot[38] instead of static image generation. Because each browser knows its own size, it can render the image size and layout accordingly for the best viewing experience. The webpage would refresh every 5 seconds, request new sensing data from the backend system, then generate the new image locally on user's personal computer, laptop, or mobile phone. The browser needs to fetch enough sensing data in each request of sensing data for image generation. When Internet is slow or unstable, the display will be delayed, but local manipulation will still be responsive. Although our second-generation system has solved the problem with static file generation of the first generation, it still suffers from high data traffic in every webpage refresh.

Our third-generation data dissemination subsystem tries to reduce the redundant data traffic pulled by the browser in each refresh. The local browser can keep a data window that caches the received data and update only the most recently sensed data from the server. Upon each data request, the browser pulls only the newly added data since the last data request. The amount of transferred data is reduced by about two orders of magnitude, from thousands of bytes to several tens of bytes in average.

In addition to pulling, we can also push newly added data from backend system to connected browsers using a new technology, websocket, introduced along with HTML5. We further improve the user experience by converting fixed-rate data pulling by the browser to data pushing by the backend server. Pushing enables the backend to control the web page content based on the incoming data rate and is more bandwidth-efficient. This also eliminates the problem of sudden massive simultaneous requests to the backend server, which can overwhelm the server. In addition to the data traffic improvement, we also create mashup services by integrating existing web servics such

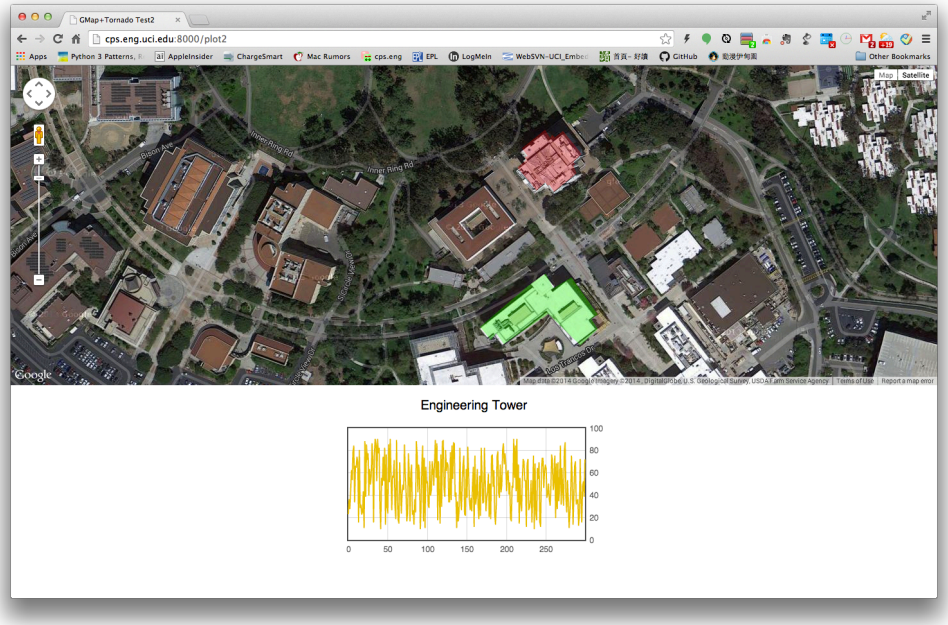


Figure 4.7: Google Map Service Integration

as Google Map[23]. We can use the latest and scalable map information instead of fixed image. A sample monitoring system is shown as Figure. 4.7.

After the above improvements in the data dissemination subsystem of our backend system, the system was upgraded from a file provider to a data provider and then to a content source provider through the Tornado web server. Now the content provided by our backend system, i.e., the acceleration data from Gophers, can be subscribe to by not only our own dynamic web pages but also other HTML5 clients in general. All these web message exchange are in JavaScript Object Notation (JSON) as a widely used format.

4.3 Evaluation

We have performed multiple short-term and long-term experiments with our PipeTECT system. Short-term experiments are mostly to validate the capability and stability of the DuraMote smart sensor system. Our test sites include

- Vincent Thomas Bridge in Long Beach, California
- Hwamyeong Bridge in South Korea
- A miniature pipeline model
- Buildings at University of California, Irvine
- Fire hydrant at a fresh water facility

Experimental results on the collected data have been presented in our previous papers. The experience from the short-term experiments provided feedback for us to further developing our backend system. We also installed our PipeTECT system in several buildings at the University of California, Irvine for our long-term experiment.

4.3.1 Lessons from Field Experiments

From the field experiments, we have learned lessons about the scalability, system stability, and network performance. The weakest links and bottlenecks have been identified in all different subsystems.

Initially, our system was not built with scalability in mind. In our short-term experiments, we deployed only several DuraMote sensors and backend system worked well, but our experiments on the Hwamyeong Bridge in Busan, South Korea, revealed our performance bottlenecks, where communication speed, bandwidth, and computation delay can all limit the number of sensing units that a backend system can serve. Each Gopher node sampled at 450Hz and generated 3.6 KB of data per second. The 802.11n Wi-Fi infrastructure provided sufficient network speed and bandwidth for the 14 Gopher nodes that generated a total of 50.4 KB of data needed to be handled per second. However, the computation delay turned out to be a performance bottleneck. The system was failing due to insufficient processing time for the networking task in our original design. It

combined networking and data processing, which took too much time to process the data and could not keep up with the incoming packets. As a result, the backend system exhausted all available memory and crashed.

In Hwamyeong Bridge deployment, we made the in-field backend system to provide collected data through web service for our centralized backend system located in University of California, Irvine to access. Although the data could be delivered to the centralized backend, the amount of data traffic was more than the cross-Pacific network bandwidth, this led to poor visual presentation performance and unstable network connections. We have overcome this bottleneck by converting to websockets.

In these short-term experiment, we stored all data as files. The file-based storage lacked an efficient query interface for later data dissemination, and we have since converted it to MySQL and SQLite database storage as our data logging engine. The SQLite database is a relative simple database compared to MySQL. Since SQLite database can be found on mobile devices, our mobile version of backend system uses SQLite for logging and local browsing at hand. MySQL is a more powerful database that provides a fast query interface. When searching for a certain time epoch of data in all stored data, it outperforms the file-based storage.

4.3.2 Data Traffic Optimization

This subsection shows quantitative results of data traffic improvement by upgrading our data dissemination task from static files to dynamic web content to websockets, as described in Section 5.5. As shown in Table 4.1, the data traffic improvement was three-fold while the image generation time reduced slightly. At the same time, the modular plug-in system implementation enabled seamlessly software upgrade. The transmitted data size has also reduced.

For the baseline, our graphical user interface generated static files using the Matplotlib Python

Table 4.1: Improvements in Data Dissemination Subsystem

System Types	Image gen. method	Image gen. time	Transmitted data
Static file	Matplotlib(Python)	0.69 s	64 KB
Dynamic web content	Flot(Javascript)	0.56 s	1.5 KB
Incremental content update	Flot(Javascript)	0.56 s	>20 B

Figure 4.8: Locations of deployed sensors

library. The backend system creates new image files as new sensing data arrive, but to avoid a surge of the image files that all need to be created within a short period, image generation process of each Gopher is separate by time. As a result, the average image generation time is 0.69 second, but the delay can easily add up to several seconds if the backend system needs to take care of more PipeTECT sensing systems. The generated image size is nontrivial and takes about 64 KB in 800 X 600 resolution. User can feel the delay when the web page is refreshing. A Javascript-based solution has been developed to solve these problems, as the image generation process is now run in the browsers for more responsiveness.

The size of transmitted data from backend to browse is almost 43 times less then before. To view the result of collected data is much smoother not but still not ready for a larger monitoring system. When more then 17 PipeTECT systems exists, the communication delay happens due to too much data need to be sent from backend system. To further improve our system, an incremental update

Figure 4.9: Engineering Tower

concept has brought in. The transmitted data is cached locally and the backend system only sent newly added data to browser. Only then the browser requests for a full set of data for display then big chunk of data is transmitted. The transmitted data is now no more than 20 bytes for a single PipeTECT system shown on the web page. We also make the incremental update only occur with viewed PipeTECT system. Third is the load on backend has been alleviated due to the image generation now moves to browsers. Backend system only is responsible of data transmission and the browser do the heavy image generation. This enables the backend to focus on data processing and leave all other display tasks to the actual viewing devices.

4.3.3 Long-Term Installation at UC Irvine

We have installed DuraMote smart sensors in several buildings and on utility pipes at the University of California, Irvine to evaluate our proposed backend system. Wi-Fi infrastructure is available for direct Internet connectivity. The proposed backend system is implemented on an Intel Pentium-4 3.0 GHz desktop PC with 4 GB of memory and installed with Ubuntu Linux operating system version 11.10. This non-server class PC is utilized to show our backend system does not required tremendous computing power.

Figure 4.10: Engineering Hall

Figure 4.11: CALIT2

Figure 4.12: Engineering Gateway

Figure 4.13: AIRB

The monitored area covered by this deployment is approximately 200 m^2. Five nodes were installed on five different buildings inside the engineering quad of UCI campus. Figure. 4.8 shows all the monitored buildings in covered area. The locations of the nodes are

- Basement of Engineering Tower
- Fourth floor of Engineering Hall
- Fourth floor of Engineering Gateway
- Roof of CalIT2 building
- Fourth floor of AIRB building

as shown in Figure. 4.9 to Figure. 4.13. The backend server is placed on the ground floor of the AIRB building, where the laboratory is located.

All DuraMote sensors are connected to the campus Wi-Fi and send collected data directly to the server in the AIRB building. Most Gopher nodes are configured for a sampling rate of 150 Hz, while the one in the basement of Engineering Tower is configured for 1000 Hz to monitor a utility pipe. The deployed nodes have been monitoring since 2012 for more then a year and a half. It successfully demonstrates that our backend system is ready for permanent installation.

During the installation, several software upgrades were necessary to improve the performance and functionality of our backend system. Due to the modular design, we were able to avoid rebooting, which would have affected the network connection of these deployed sensor and disrupt the monitoring process.

Chapter 5

BlueRim: Rimware for Enabling Cloud Integration of Networks of Bluetooth Smart Devices

5.1 Related Work

Cloud support for IoT systems can be divided into domain-specific and general-purpose systems. As cyber-physical systems (CPS), different IoT-Cloud combinations may emphasize more cyber or more physical aspects. We also review related work on application-building and management tools for edge nodes of IoT.

5.1.1 Cloud Systems

Domain-Specific IoT Cloud

Cloud systems have been supporting IoT applications in domain-specific ways. In particular, sports and fitness devices such as heart-rate monitors, pedometers, and wireless personal-area network (WPAN) products connect to smartphones that upload data to the user account in the domain-specific cloud. The uploaded data can be composed with other services including social networks and map services for the users to share their sports activity updates with their friends. These edge nodes perform minimal or no access control, and the cloud supports domain-specific data management but no device management.

General-Purpose Cloud Systems

Besides the domain-specific cloud, which is usually provided as *Software as a Service* (SaaS), other types of cloud systems serve general computing purposes. *Infrastructure as a Service* (IaaS), such as Amazon EC2, Microsoft Azure, and OpenStack, offers virtual machines together with physical computing resources such as CPU, memory, and disk storage for users to deploy applications the same way as on their local machines. *Platform as a Service* (PaaS), such as Openshift, CloudFoundry, and Heroku, allow users to choose the platform components including the operating system, programming language runtime environment, web server, and database engine for building their online/offline application while automatically managing the underlying computing and storage resources for availability and scalability.

5.1.2 HTTP Servers for Sensor Networks

Kurschl 2009[51] set up a computer as a gateway and HTTP server on behalf of a network of nodes. In contrast, Rajesh et all 2010[74] assumes powerful nodes that are capable of HTTP, rather than relying on a gateway node. Similarly, Sensor-Cloud Computing[56] assumes smartphones as sensor nodes, which can connect directly to the Internet without requiring a gateway, but these nodes may be too powerful or costly to serve as IoT edge nodes. In either case, HTTP alone primarily serves as a transport rather supporting data handling and management, but they do not address access control or security concerns in sensor-cloud systems[72].

5.1.3 Application-Building Tools

At the lower level, TI's BTool[36] is a Windows program that works as a protocol analyzer: it can snoop, parse, extract BLE packets, and form and transmit packets interactively by following the profiles. BTool can emulate a BLE master or slave node before it is ready. LightBlue [73] is an iOS app that serves a similar purpose to BTool in that it understands profiles and enables users to send and receive BLE packets as defined by profiles. They are powerful for building applications on the physical (sensors/actuators) side, but they do not support Internet connectivity and cloud integration.

5.1.4 Device Management

In IoT, nodes can be dynamically added, removed, or replaced in the system with the same type of device or a different type but similar capability. Whole or partial image re-programming at runtime is needed to configure the node for replacement [43, 32]. A sensor profile service can configure newly added nodes and provide compatible data [77].

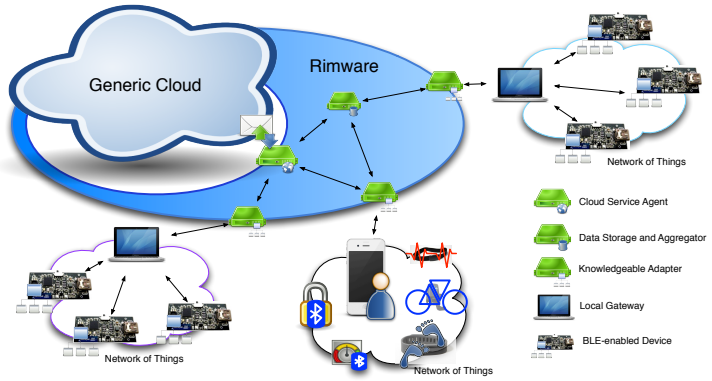


Figure 5.1: Relationship between Generic Cloud, Rimware and Network of Things

5.2 Concepts

This section explains concepts of rimware for IoT-cloud integration as shown in Figure. 5.1. The term *network of things* (NoT) refers to a locally connected network of nodes with (sensing, actuation) interfaces to the physical world, whereas the *Internet of Things* (IoT) refers to the interconnected NoTs bridged by the Internet Protocol (IP). We first provide a background on BLE for NoT (local device-to-device) communication, which can operate without rimware. Second, we explain how a *security domain* can be defined by associating NoTs with a slice of rimware. Third, we show rimware serving as the delegate of any NoT to different cloud domains.

5.2.1 A Basic BLE Network of Things

The services of a BLE device are represented as a *profile*, which consists of a collection of related *characteristics* that represent the functionality of that service in terms of one or more *descriptors* in a human-readable way. To communicate with a BLE device without hardwired knowledge, a host application must discover and load the device's services and characteristics dynamically. Our first approach is to build the knowledge layer in rimware based on different types of BLE devices and separate the knowledge layer from the host application, as described in Section 5.3.

Our second approach, as described in Section 5.4.1, is to enhance the descriptors in BLE to capture this knowledge without relying on the rimware to maintain the knowledge base. We describe our auto-adaptive way for building a host application with or without the knowledge layer for BLE devices, where in rimware the host application works as the uplink/downlink gateway for the NoT.

5.2.2 Rimware with Single Security Domain

A *security domain* is defined by a set of networked devices and applications that agree to use a common security token for authentication, authorization, or session management [89]. In NoT, there may be no strict security-checking phase by default upon connecting and pairing with devices, except for possibly simple PIN input in most situations. This means anyone can potentially *connect* to devices via its NoT-level protocol, even if one is not permitted to *access* its services. Rimware serves as the agent between any user and NoT devices and checks the user's access privilege before the user can access the devices. In addition, rimware secures the communication when a user is accessing any NoT device through rimware. Moreover, rimware also takes care of the security checking when a user is accessing any NoT device through NoT-level protocol without the connectivity to the rimware through Internet.

5.2.3 Rimware with Multiple Security Domains

Rimware can integrate multiple security domains that the users and NoT devices may belong to. Although users and devices may have unique IDs within their own domains, these IDs may not be universally unique across domains unless combined with the domain ID somehow. However, users may not want to give out their global identifier because it compromises their privacy and exposes them to identity theft and other risks. For these reasons, we proposed the concepts of *identifier* and *cyptonym* in rimware to support better privacy.

An *identifier* is a string of data for addressing an entity within a name space. Four types of identifier are (1) A *device identifier* d^i and (2) a *user identifier* u^j are those of device i and user j in some domain D^k they respectively belong to. (3) a *domain identifier* is that of a security domain, and (4) a *global identifier* is one that can address a a user, a device, a domain, or any entity within or across domains that are integrated by rimware. Device and user identifiers can be made global by composing with the domain identifier, such as (D^k, d^i) and (D^k, u^j). While all these *identifiers* are in clear text with variable length, rimware fixes the length of *domain identifier* and *global identifier* using hash function before exchanging them within rimware as well as across different domains.

Sharing identity across domains can also be described by *identifier* as a tuple. For example, sharing the identity of device i from domain j to user k from domain m can be described (g_i^j, g_k^m), where g_i^j, g_k^m are the global identifiers of the user and device, respectively. To protect privacy, the tuple of shared identity is encrypted using an irreversible hash function. We call the hashed tuple as *cyptonym*. The *cyptonym* guarantees that user or device's identity will not be exposed outside their own domains.

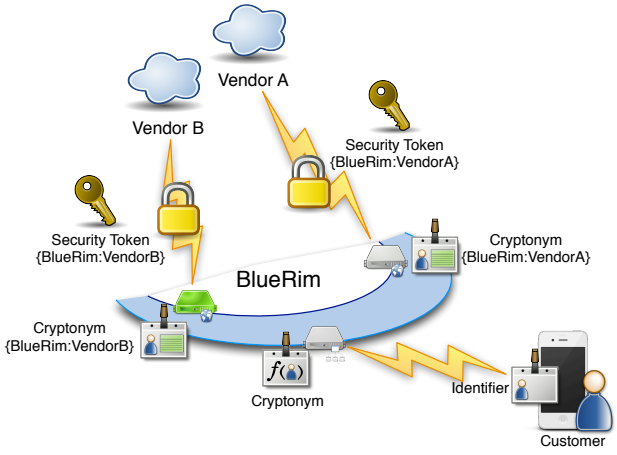


Figure 5.2: Customer, BlueRim and Vendors

An example in Figure. 5.2 that can take advantage of cryptonyms is active tracking of consumer shopping behavior using BLE tags when approaching vending machines or shops. Instead of giving out one single identifier to every vendor, a safer option is to give out a different cryptonym to each

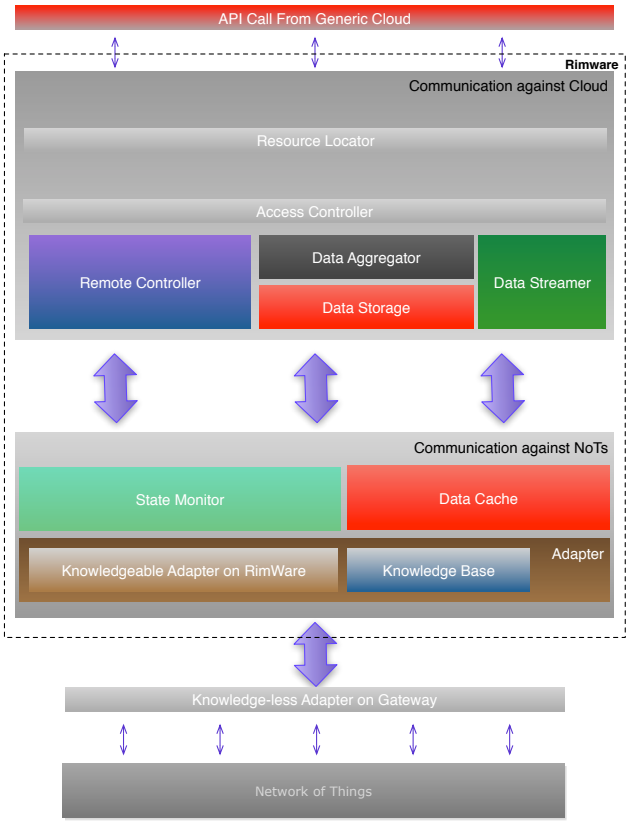


Figure 5.3: Components in Rimware

vendor (security domain), as one cryptonym is meaningless in the context of another security domain. In case of tag replacement, the consumer's rimware can re-associate the new hardware with the consumer. Additional enforcements rules, such as at most one tag may be associated with a particular user, may also be enforced.

The next two sections provide more details of our rimware design with BlueRim and its use in bridging BLE-enabled NoTs and cloud services.

5.3 BlueRim: Cloud and Rimware

Our approach to cloud for IoT is the separation between rimware and the generic cloud that provides much of the computation and storage functionality. Rimware, however, is more than a

"pass-through" of functionality to the cloud: it duplicates core cloud functionality to enable the NoT to continue operating even during loss of Internet connectivity. This section first describes these cloud-like components in the rimware, followed by a description of how these components in rimware can be replicated and distributed to integrate with different security domains and to provide high availability. The overview of components in rimware is as shown in Figure. 5.3.

5.3.1 Cloud Components in Rimware

The cloud-like components in rimware present a service abstraction for not only functionality on NoT devices but also any static resources such as logged data from these devices.

Knowledge Base and Knowledgeable Adapter

Rimware can serve as the *knowledge base* for mapping between the *abstract actions* (i.e., what the user wants to achieve) and the concrete *action commands* (i.e., the actual message formats needed). Each instance of the mappings as maintained by each registered device in the rimware is called a *knowledgeable adapter*. The mappings include standard ones defined by Bluetooth SIG as well as custom ones defined by developers, and more mappings can be added later by either the device vendor or the administrator systematically.

When a gateway program discovers a nearby BLE device, it collects and submits the device's service/characteristic profile to BlueRim. From matching in the knowledge base, BlueRim recognizes the device type and creates a knowledgeable adapter for it. The knowledgeable adapter can detect changes to the firmware specification by making adjustments accordingly by consulting the knowledge base. The knowledge base also aids crash recovery and state restoration. It keeps the firmware image of any registered device as well as what state should be monitored. In case of a crash that requires hardware replacement, BlueRim can easily restore the image by retrieving it

from the knowledge base.

BLE allows the functionality of any characteristic profile to be stated in a human-readable way using *descriptors*. We plan to define a structured way to make descriptors whose semantics can be further utilized to assist with maintaining the knowledge base without the participation of device vendors or administrators.

State Monitor

The *state monitor* in rimware consists of a collection of active tables that monitor the state of those NoTs registered in the rimware. The state monitor take snapshots of the state on both the gateways and devices by periodically sending pull requests to all gateways. The state table is indexed by both the gateway identifier and device identifier. The state to monitor on the gateways includes on/off state, the list of connected devices (in terms of their identifiers), geo-location, etc. The state to monitor on the devices depends on how the type of the device is modeled in the knowledgeable base. Besides providing the state information to the users, the state monitor can also *restore* a device to its latest snapshot of state when the device has crashed and must be reset.

Data Cache

A *data cache* in rimware is an in-memory cache. It caches the data transmitted from the gateways before putting them into the data storage for archive. It also serves as the engine of any real-time monitor request from the end user against any device(s). The data cache keeps only the latest data not yet written to the data storage or not pushed to real-time request.

Remote Controller

As the representation layer of the knowledgeable adapter, the *remote controller* in rimware receives the user's action request and sends it to the knowledgeable adapter to take action on the corresponding device. Similar to a knowledgeable adapter, a remote controller has a 1-to-1 relationship with each registered device.

Data Storage and Data Aggregator

The *data storage* in rimware is for archiving short- and long-term logged data from any device. Data in *data storage* is normally stored in relational database tables for general user queries. For queries against large amounts of historical data for analysis purposes, we use a *data aggregator* to pre-process the data from an Online Analytics Processing (OLAP) engine [35]. A data aggregator is a data warehouse where data in relational tables are moved and re-organized into a multi-dimensional structure according to the dimension attributes such as time, geo-location, device type, etc. Data in different dimensions are pre-aggregated hierarchically. For example, in time dimension, data can be aggregated according to month, quarter, or year. These pre-aggregating rules are defined by the system, and new rules can be added from data analytics.

Data Streamer

A *data streamer* is the representation layer of the data cache. It takes the user's real-time monitoring request and returns the latest data received from the device. The data streamer is 1-to-1 related to the user's request for real-time streaming.

5.3.2 Integration of Rimware with Multiple Security Domains

Rimware also can integrate multiple security domains with Access Controller and Resource Locator components.

Access Controller

The access controller is to interpret privileges for a local NoT device to a cloud service. When a user accesses a device locally through the NoT gateway, his/her privileges are recognized by the local gateway. Rimware bridges between different NoTs and also with cloud service. The user retains the same privileges when accessing through rimware.

For example, consider an NoT with two accounts: guest and administrator. The administrator can change parameters of a sensor in the device and view the data, but a guest is allowed to view collected data only. A user from the cloud side will be mapped to either a guest or an administrator to the NoT.

Resource Locator

Resource locator is to resolve the communication path between user and targeted devices with given *tags*. A tag is a user named label which is associated with cryptonyms mentioned in Section 5.2. In rimware, an action is represented in a URL-like format using our own protocol in the form of `rim://tag/action/action_type`. When a user issues actions, the resource locator determines the cryptonyms and communication path, and the knowledgeable adapter converts the actions into pure BLE commands and asks the corresponding gateway to send the commands to the device. Without resource locator, a user can communicate with the targeted devices by using cryptonyms.

5.4 BlueRim: Physical Subsystem

The physical subsystem of the BlueRim can be divided into two parts. The first part is the gateway, which bridges the local NoT and the Internet. The second part consists of the nodes in local NoTs. To assist developers with building up a BLE NoT to be integrated with our BlueRim, we provide a host-program builder. For the firmware on the nodes, some required features need to be implemented on the nodes to work with BlueRim.

5.4.1 Host-Program Builder

Our host-program builder currently targets BLE. On the firmware side, operations on a BLE device are essentially read/write against the device's characteristic profiles. An abstract action, e.g., turning on the light, may involve one or a series of read/write operations against different characteristics on a lighting-control device. To control a BLE device, the same structure of service/characteristic profiles should be established in the host program so that it can map to the structure on the firmware side and send or receive the respective action commands. The host program builder can help build the mapping in two ways: by scanning profile information on a connected BLE device or from user's direct input. The host-program builder provides three basic interfaces for each characteristic: *read*, *write*, and *onDataReceived*. The *onDataReceived* function is a callback function used to catch events caused by either a *notification* from the device side or a return of a read request sent from the host side. Through provided interfaces, developers can specify action commands against characteristics. It helps developers further build a knowledge layer by creating mappings between abstract actions and action commands.

As we mentioned in Section 5.3, in our design, the knowledge layer is taken care of by rimware, which does not exist on gateway program. The gateway program only adapts to the newly connected BLE device by establishing a service/characteristic structure for sending action commands

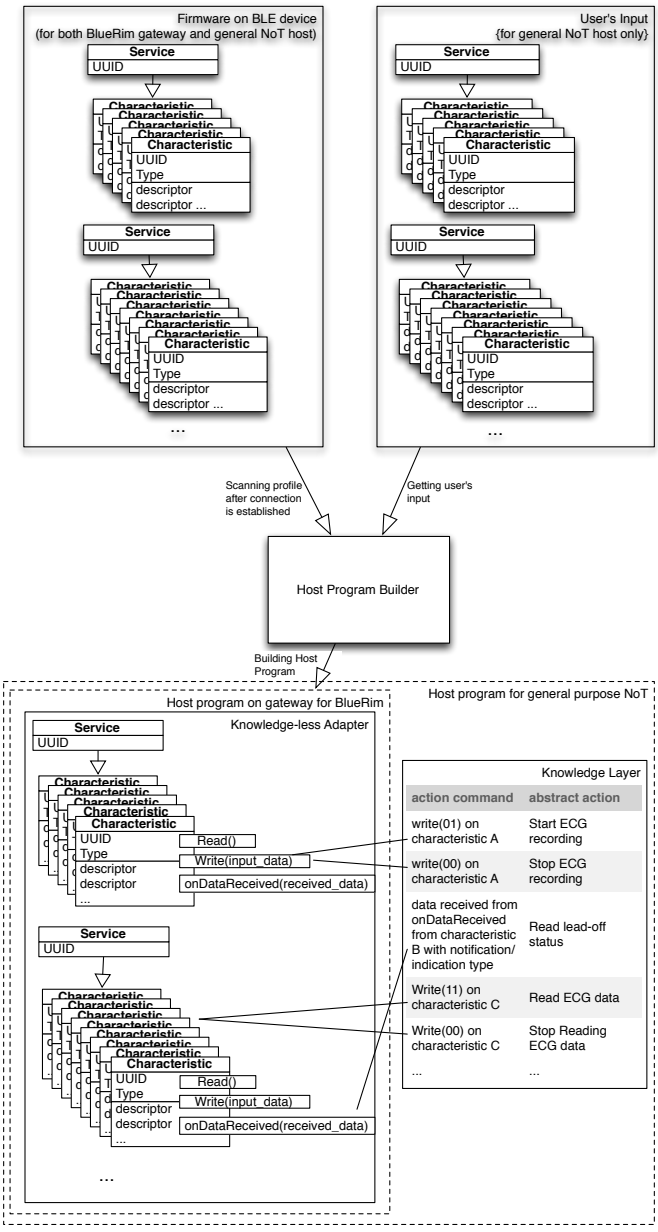


Figure 5.4: Workflow of Host Program Builder

to the device. This process is done automatically. The workflow of the host-program builder for building BlueRim gateways and general-purpose NoT host programs is shown in Figure. 5.4.

5.4.2 NoT Firmware Support

The BLE nodes in the NoT also follow the service/characteristic profile structure. The communication is done with characteristic read/write operations. A BlueRim service profile is needed in the NoT firmware. This profile contains the characteristics to hold BlueRim-assigned global identifier and provides device signatures for the identifier-generation step. When there is no Internet connectivity, edge gateways to BlueRim can cache collected data from connected NoT devices using global identifier. This profile support enables the BlueRim edge gateway to update its data cache periodically instead of requiring constant connection.

5.5 Case Studies

5.5.1 Infant ECG

One of our driving examples is an infant ECG (electrocardiogram) for studying the long-QT syndrome. It includes a pre-positioned flexible electrode strip placeable by minimally trained parents on infants to digitize and record ECG signals onto a flash memory card. Afterwards, it uploads the data wirelessly via a BLE-enabled smartmobile or PC to the server.

BlueRim acts as the data repository, user interface, user management, and device administration tool. It enforces access privileges for the parents, doctors, and administrators of the data and ECG devices, and it supports device configuration (data upload, erase, ID setting, sampling rate setting, firmware update) automatically constructed using our custom BLE profiles. For a small-scale

study, a single machine with an Internet connection may be sufficient to run BlueRim. Additional machines at different sites can be added to the same security domain. BlueRim also supports plug-ins for ECG data filtering, segmentation by heart beat, QT analysis, and running statistics on the analyzed data. With BlueRim, these features only need to be enabled and parameterized rather than re-implemented, making it very quick to and reliable to build a new application.

5.5.2 Home Automation

Our home automation application involves light switches and appliance outlets whose on/off action may be triggered by a physical button on the wall, by an occupancy sensor wirelessly connected to the switch, a button that is wirelessly connected to multiple switches (e.g., upstairs/downstairs switches to control the same light), a smartphone, or a computer. In addition, we also support over-the-Internet status observation, on/off control, and integration with cloud services such as Google Calendar. BLE as the local-area wireless protocol has the advantage of direct compatibility with smartphones such as iPhone 4S and later as well as Windows 7.

The main issues highlighted by this application include robustness to central point of failure, access control, and multiple security domains. First, the light switches and appliances should continue to work from local control (button press or smartphone control) even if the gateway loses connectivity or fails. Second, only certain users are allowed to control a subset of the lights or even observe their status. For example, adjacent offices may belong to the same security domain but may not to share control; Adjacent apartments may actually belong to different security domains even if their RF ranges overlap. In contrast, with most WiFi-based remote controls, anyone who can join the WLAN often gains unlimited access to the system.

Chapter 6

Conclusion

In this work, we have propose various runtime supports from edge devices to cloud services. For edge devices, while modern intergraded RF-MCUs, we first propose Enix makes several contributions in the WSN OS area. First, the cooperative threads programming model enhances the performance by decreasing context-switch overhead, making it two times faster than the traditional preemptive multi-threaded programming model. This cooperative threading model is easy to learn compared to the event-driven model. Moreover, the local states can be saved and restored automatically during context switches without burdening the programmer with manual state saving in some other programming models. Second, Enix provides code virtual memory to overcome the shortage of on-chip code memory via host-assisted demand segmentation, which most WSN OSs do not support. To achieve virtual memory, the ELIB is built on the host PC composed of PIC segments and is loaded to code memory on-demand by the run-time loader of Enix. Third, remote reprogramming is also available in Enix. Due to the pre-stored ELIB on the Micro-SD card, the size of the binary image of the user program to be wirelessly updated is reduced significantly during the remote reprogramming stage. Finally, the code and data footprints of Enix with full-function including EcoFS are at most 8KB and 512 bytes, respectively, which are the smallest compared to other WSN OSs. Only ten percent of code is machine-dependent, while the rest is

written in C language, and thus it is easy to port to other wireless sensor platforms.

While BLE are getting more popular, vendor provided schedulor with BLE stack is used. We also develop a dispatcher on top of existing vendor scheduler that we can use vendor supported communication stack and build a unified dispatching framework.

As for runtime support in the gateway devices, we bring the elegance and power of Python to enabling interactive programming of resource-constrained wireless sensor networks. Interactivity encourages experimentation by beginners, by providing instant feedback to functions of interest while being forgiving by supporting undo with an underlying revision control system. As Python is not just a "beginner's language" but is actually used by expert programmers in production-quality systems due to support of constructs at a much higher-level of abstraction, we expect EcoCast to also boost the productivity of expert programmers in a similar way.

While BLE has impacted the Network-Of-Things, based on EcoCast, PyBLE is developed to present any existing pheripherals. PyBLE provides constrcuts for users to build their own periphrals from basic profiles and characteristics. With BLE, it enables user to focus on data processing instead of data communications.

Another important functionality of these gateway devices is that they can perform some basic data processing. We describe our modular design of a backend computing system for continuous monitoring of civil structures and water pipelines. Its use of the Python programming language enables different software components to be developed in a modular way, and the kernel system loads the required modules accordingly using the internal plugin system. The downtime of the backend system is minimized since the software module loading is seamless without affecting the rest of the system. Based on our experience with deploying our smart sensors, we improved our backend system by enhancing the data logging facilities with fast data query. and optimizing the data display for reduced network traffic by two orders of magnitude, making the backend system much more scalable. We also create mashup services by integrating Google Map overlaid

on our data visualizer. One direction for future work is to integrate even more services such as earthquake and weather information to enable cross references of collected data for discovering more knowledge about these structures being monitored.

Finally, to bridge the cyber and the physical world, base on runtime supports we developed from in edge and gateway devices. We have introduced the concept of rimware for the integration of the Internet of things and the cloud to form a very powerful cyber-physical system. Our specific implementation, called BlueRim, can scale from isolated networks of things (NoT) to the global IoT. BlueRim provides services for systematic data handling and dissemination, device management and access control, and device roaming within and across multiple security domains. In addition, we provide an application builder that exploits Bluetooth Smart's attribute profiles to enable systematic construction of new IoT applications. The effectiveness of these fundamental features have been validated in several real-world applications that with different access patterns while retaining their ability to consume very low power. We believe that our approach represents an important technology in taking IoT closer to realizing the full potential.

Bibliography

[1] MicaZ. http://www.xbow.com/Products/productdetails.aspx?sid=164.

[2] A. Adya, J. Howell, M. Theimer, W. J. Bolosky, and J. R. Douceur. Cooperative task management without manual stack management. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference*, pages 289–302, Berkeley, CA, USA, 2002. USENIX Association.

[3] S. H. Amer and J. A. Hamilton, Jr. DSR and TORA in fixed and mobile wireless networks. In *ACM-SE 46: Proceedings of the 46th Annual Southeast Regional Conference*, pages 458–463, New York, NY, USA, 2008. ACM.

[4] L. S. Bai, L. Yang, and R. P. Dick. Automated compile-time and run-time techniques to increase usable memory in MMU-less embedded systems. In *CASES '06: Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 125–135, New York, NY, USA, 2006. ACM.

[5] L. S. Bai, L. Yang, and R. P. Dick. MEMMU: Memory expansion for MMU-less embedded systems. *ACM Trans. Embed. Comput. Syst.*, 8(3):1–33, 2009.

[6] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson, and R. Han. Mantis os: an embedded multithreaded operating system for wireless micro sensor platforms. *Mob. Netw. Appl.*, 10(4):563–579, 2005.

[7] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *Mobile Data Management*, pages 3–14. Springer, 2001.

[8] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *The First International Conference on Mobile Systems, Applications, and Services (MobiSys)*, May 2003.

[9] J. Broch, D. A. Maltz, D. B. Johnson, Y.-C. Hu, and J. Jetcheva. A performance comparison of multi-hop wireless ad hoc network routing protocols. In *MobiCom '98: Proceedings of the 4th annual ACM/IEEE international conference on Mobile computing and networking*, pages 85–97, New York, NY, USA, 1998. ACM.

[10] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He. The LiteOS operating system: Towards unix-like abstractions for wireless sensor networks. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks*, pages 233–244, Washington, DC, USA, 2008. IEEE Computer Society.

[11] H. Cha, S. Choi, I. Jung, H. Kim, H. Shin, J. Yoo, and C. Yoon. RETOS: resilient, expandable, and threaded operating system for wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 148–157, New York, NY, USA, 2007. ACM.

[12] C. Chen, Y. Chen, Y. Tu, S. Yang, and P. Chou. EcoSpire: an application development kit for an Ultra-Compact wireless sensing system. *Embedded Systems Letters, IEEE*, 1(3):65–68, 2009.

[13] S. Choudhuri and T. Givargis. Software virtual memory management for MMU-less embedded systems. Technical report, Center for Embedded Computer Systems, University of California, Irvine, Nov 2005.

[14] T. Clausen and P. Jacquet. RFC3626: Optimized Link State Routing Protocol (OLSR). *RFC Editor United States*, 2003.

[15] Crossbow Technology. Mote in-network programming user reference version 20030315. `http://www.tinyos.net/tinyos-1.x/doc/Xnp.pdf`, 2003.

[16] H. Dai, M. Neufeld, and R. Han. ELF: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04*, pages 176–187, New York, NY, USA, 2004. ACM.

[17] C. Duffy, U. Roedig, J. Herbert, and C. J. Sreenan. Adding preemption to TinyOS. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors*, pages 88–92, New York, NY, USA, 2007. ACM.

[18] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06*, pages 15–28, New York, NY, USA, 2006. ACM.

[19] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 15–28, New York, NY, USA, 2006. ACM.

[20] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[21] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06*, pages 29–42, New York, NY, USA, 2006. ACM.

[22] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[23] I. Google. Google maps apis.

[24] L. Gu and J. A. Stankovic. t-kernel: providing reliable os support to wireless sensor networks. In *SenSys '06*, pages 1–14, New York, NY, USA, 2006. ACM.

[25] A. Gustafsson. Threads without the pain. *Queue*, 3(9):34–41, 2005.

[26] J. Hahn, Q. Xie, and P. H. Chou. Rappit: framework for synthesis of host-assisted scripting engines for adaptive embedded systems. In *(CODES+ISSS'05)*, pages 315–320, September 2005.

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

[28] W. R. Heinzelman, A. Chandrakasan, and H. Balakrishnan. Energy-efficient communication protocol for wireless microsensor networks. In *HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8020, Washington, DC, USA, 2000. IEEE Computer Society.

[29] J. Hill and D. Culler. Mica: a wireless platform for deeply embedded networks. volume 22, pages 12–24, 2002.

[30] D.-D. Ho, P.-Y. Lee, K.-D. Nguyen, D.-S. Hong, S.-Y. Lee, J.-T. Kim, S.-W. Shin, C.-B. Yun, and M. Shinozuka. Solar-powered multi-scale sensor node on lmote 2 platform for hybrid shm in cable-stayed bridge. *Smart Structures and Systems*, 9(2):145–164, 2012.

[31] C.-H. Hsueh. EcoExec: A highly interactive execution framework for ultra compact wireless sensor nodes. Master's thesis, National Tsing Hua University, 2009.

[32] C.-H. Hsueh, Y.-H. Tu, Y.-C. Li, and P. Chou. EcoExec: An Interactive Execution Framework for Ultra Compact Wireless Sensor Nodes. pages 1 –9, jun. 2010.

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

[34] J. D. Hunter. Matplotlib: A 2d graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[35] S. Ikeda, P. C.-Y. Sheu, and J. P. Tsai. A Model for Object Relational OLAP. *International Journal on Artificial Intelligence Tools*, pages 551–595, 2010.

[36] T. I. Incorporated. BTool from TI's BLE stack.

[37] Intel. Portable Formats Specification, Version 1.1.

[38] IOLA and O. Laursen. Flot: Attractive javascript plotting for jquery.

[39] J. Jeong and D. Culler. Incremental network programming for wireless sensors. In *Sensor and Ad Hoc Communications and Networks, 2004. IEEE SECON 2004. 2004 First Annual IEEE Communications Society Conference on*, pages 25–33, Oct. 2004.

[40] M. Jiang, J. Li, and Y. Tay. *Cluster based routing protocol (CBRP)*. IETF MANET Working Group, Internet-Draft, August 1999.

[41] D. B. Johnson and D. A. Maltz. Dynamic source routing in ad hoc wireless networks. In *Mobile Computing*, pages 153–181. Kluwer Academic Publishers, 1996.

[42] J. Kim and P. H. Chou. Remote progressive firmware update for flash-based networked embedded systems. In *Proc. International Symposium on Low Power Electronics andDesign (ISLPED)*, pages 407–412, San Francisco, CA, USA, August 19-21 2009.

[43] J. Kim and P. H. Chou. Remote progressive firmware update for flash-based networked embedded systems. In *Proceedings of the 14th ACM/IEEE international symposium on Low power electronics and design*, pages 407–412, San Fancisco, CA, USA, 2009. ACM.

[44] S. Kim, M. Torbol, and P. H. Chou. Remote structural health monitoring systems for next generation scada. *Smart Structures and Systems*, 11(5):511–531, 2013.

[45] S. Kim, E. Yoon, T.-C. Chien, H. Mustafa, P. H. Chou, and M. Shinozuka. Smart wireless sensor system for lifeline health monitoring under a disaster event, 2011.

[46] K. Klues, C.-J. M. Liang, J. Paek, R. Musăloiu-E, P. Levis, A. Terzis, and R. Govindan. TOSThreads: thread-safe and non-invasive preemption in TinyOS. In *SenSys '09*, pages 127–140, New York, NY, USA, 2009. ACM.

[47] D. Knuth. *Fundamental Algorithms, Third Edition.*, chapter Section 1.4.2: Coroutines, pages 193–200. Addison-Wesley, 1997.

[48] J. Koshy. Remote incremental linking for energy-efficient reprogramming of sensor networks. In *Proceedings of the second European Workshop on Wireless Sensor Networks*, pages 354–365. IEEE Press, 2005.

[49] J. Koshy and R. Pandey. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys '05*, pages 243–254, New York, NY, USA, 2005. ACM.

[50] S. S. Kulkarni and L. Wang. Mnp: Multihop network reprogramming service for sensor networks. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS*, pages 7–16, 2005.

[51] W. Kurschl and W. Beer. Combining cloud computing and wireless sensor networks. In *Proceedings of the 11th International Conference on Information Integration and Web-based Applications & Services*, iiWAS '09, pages 512–518, New York, NY, USA, 2009. ACM.

[52] J. J. Labrosse. *MicroC/OS-II, The Real-Time Kernel 2ND EDITION*. CMP Books, 2002.

[53] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *ASPLOS-X: Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 85–95, New York, NY, USA, 2002. ACM.

[54] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for sensor networks. *Ambient Intelligence*, pages 115–148, 2005.

[55] J. Lifton, D. Seetharam, M. Broxton, and J. Paradiso. Pushpin computing system overview: A platform for distributed, embedded, ubiquitous sensor networks. In *Pervasive*, pages 139–151, 2002.

[56] Y.-H. Liu, K.-L. Ong, and A. Goscinski. Sensor-Cloud Computing: Novel Applications and Research Problems. In R. Benlamri, editor, *Networked Digital Technologies*, volume 294 of *Communications in Computer and Information Science*, pages 475–486. Springer Berlin Heidelberg, 2012.

[57] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: an acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems (TODS)*, 30(1):173, 2005.

[58] P. J. Marron, M. Gauger, A. Lachenmann, D. Minder, O. Saukh, and K. Rothermel. FlexCup: A flexible and efficient code update mechanism for sensor networks. In *EWSN '06: Proceedings of the third European Workshop on Wireless Sensor Networks (EWSN 2006*, pages 212–227, 2006.

[59] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: an energy-optimized object storage system for memory-constrained sensor devices. In *SenSys '06*, pages 195–208, New York, NY, USA, 2006. ACM.

[60] R. Müller, G. Alonso, and D. Kossmann. A virtual machine for sensor networks. In *EuroSys '07: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*, pages 145–158, New York, NY, USA, 2007. ACM.

[61] S. Murthy and J. J. Garcia-Luna-Aceves. An efficient routing protocol for wireless networks. *Mob. Netw. Appl.*, 1(2):183–197, 1996.

[62] S. Nath and A. Kansal. FlashDB: dynamic self-tuning database for NAND flash. In *IPSN '07*, pages 410–419, New York, NY, USA, 2007. ACM.

[63] R. Ogier, F. Templin, and M. Lewis. Topology dissemination based on reverse-path forwarding (TBRPF), February 2004.

[64] J. K. Ousterhout. Tcl: An embeddable command language. In *Proceedings of the USENIX Winter 1990 Technical Conference*, Berkeley, CA, 1990. USENIX Association.

[65] J. K. Ousterhout. Scripting: Higher level programming for the 21st century. *IEEE Computer*, 31(3):23–30, March 1998.

[66] S. N. Pakzad, G. L. Fenves, S. Kim, and D. E. Culler. Design and implementation of scalable wireless sensor network for structural monitoring. *Journal of Infrastructure Systems*, 14(1):89–101, 2008.

[67] C. Park, J. Lim, K. Kwon, J. Lee, and S. L. Min. Compiler-assisted demand paging for embedded systems with flash memory. In *EMSOFT '04*, pages 114–124, New York, NY, USA, 2004. ACM.

[68] C. Park, J. Liu, and P. H. Chou. Eco: an ultra-compact low-power wireless sensor node for real-time motion monitoring. In *IPSN '05*, page 54, Piscataway, NJ, USA, 2005. IEEE Press.

[69] V. D. Park and M. S. Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *INFOCOM '97: Proceedings of the INFOCOM '97. Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution*, page 1405, Washington, DC, USA, 1997. IEEE Computer Society.

[70] C. E. Perkins and P. Bhagwat. Highly dynamic destination-sequenced distance-vector routing (DSDV) for mobile computers. In *SIGCOMM '94: Proceedings of the conference on Communications architectures, protocols and applications*, pages 234–244, New York, NY, USA, 1994. ACM.

[71] C. E. Perkins and E. M. Royer. Ad-hoc on-demand distance vector routing. *Mobile Computing Systems and Applications, IEEE Workshop on*, 0:90, 1999.

[72] N. Poolsappasit, V. Kumar, S. Madria, and S. Chellappan. Challenges in Secure Sensor-Cloud Computing. In W. Jonker and M. Petković, editors, *Secure Data Management*, volume 6933 of *Lecture Notes in Computer Science*, pages 70–84. Springer Berlin Heidelberg, 2011.

[73] L. Punch Through Design. LightBlue.

[74] V. Rajesh, J. M. Gnanasekar, R. S. Ponmagal, and P. Anbalagan. Integration of Wireless Sensor Network with Cloud. In *Recent Trends in Information, Telecommunication and Computing (ITC), 2010 International Conference on*, pages 321–323, 2010.

[75] Real Time Engineers, Ltd. FreeRTOS: Free, portable, open source, royalty free, mini real time kernel. `http://www.freertos.org/`, 2010.

[76] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *WSNA '03: Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 60–67, New York, NY, USA, 2003. ACM.

[77] N. Reijers, K.-J. Lin, Y.-C. Wang, C.-S. Shih, and J. Y. Hsu. Design of an intelligent middleware for flexible sensor configuration in M2M systems. SENSORNETS, 2013.

[78] M. Samuel, J. Michael, M. Joseph, and H. Wei. Tag: A tiny aggregation service for ad-hoc sensor networks. In *OSDI*, volume 2, pages 9–11, 2002.

[79] M. Shinozuka, S. Lee, S. Kim, and P. H. Chou. Lessons from two field tests on pipeline damage detection using acceleration measurement, 2011.

[80] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical report, UCLA, Los Angeles, CA, USA, 2003.

[81] Talin. Metaclasses in python 3000, March 2007.

[82] Tidorum Ltd. Bound-T user guide. `http://www.tidorum.fi/bound-t/manuals/user-guide.pdf`, April 2009.

[83] A. Tridgell. *Efficient Algorithms for Sorting and Synchronization*. PhD thesis, Australian National University, 2000.

[84] N. Tsiftes, A. Dunkels, Z. He, and T. Voigt. Enabling large-scale storage in sensor networks with the coffee file system. In *IPSN'09*, pages 349–360, Washington, DC, USA, 2009. IEEE Computer Society.

[85] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea (for high-concurrency servers). In *HOTOS'03: Proceedings of the 9th conference on Hot Topics in Operating Systems*, pages 4–4, Berkeley, CA, USA, 2003. USENIX Association.

[86] Q. Xie, J. Liu, and P. H. Chou. Tapper: A lightweight scripting engine for highly constrained wireless sensor nodes. In *Proceedings of the fifth international conference on Information processing in sensor networks(IPSN '06)*, April 2006.

[87] S. Yamashita, T. Shimura, K. Aiki, K. Ara, Y. Ogata, I. Shimokawa, T. Tanaka, H. Kuriyama, K. Shimada, and K. Yano. A 15 $\times$ 15 mm, 1 μA, reliable sensor-net module: enabling application-specific nodes. In *IPSN '06*, pages 383–390, New York, NY, USA, 2006. ACM.

[88] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD record*, 31(3):9–18, 2002.

[89] J. Zao, L. Sanchez, M. Condell, C. Lynn, M. Fredette, P. Helinek, P. Krishnan, A. Jackson, D. Mankins, M. Shepard, and S. Kent. Domain based internet security policy management. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, volume 1, pages 41–53 vol.1, 2000.

[90] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: an efficient index structure for fash-based sensor devices. In *FAST'05: Proceedings of the 4th conference on USENIX Conference on File and Storage Technologies*, pages 3–3, Berkeley, CA, USA, 2005. USENIX Association.