ENVIROSUITE: AN ENVIRONMENTALLY-IMMERSIVE PROGRAMMING FRAMEWORK FOR WIRELESS SENSOR NETWORKS

BY

LIQIAN LUO

B.E., Tsinghua University, 2000
M.C.S., University of Virginia, 2004

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2007

Urbana, Illinois

Abstract

Networked, embedded sensors allow for an instrumentation of the physical world at unprecedented granularities and from unimagined perspectives. The advent of a ubiquitous sensing era is evident. Yet, sensor network techniques are still far from entering mainstream adoption due to multiple unresolved research challenges, especially due to the high development cost of sensor network applications. Therefore, in this dissertation, we propose to design, implement, and evaluate an environmentally-immersive programming framework, called *EnviroSuite*, to reduce sensor network software development cost. The goal of our research is to create reusable sensor network development support for the community and reduce the adoption barriers for a broader category of users, ultimately leading to a transition of sensor networks from a research concept to a general-purpose technology available for use for a wide variety of research, government, industry, and everyday purposes.

Current sensor network programming practice remains very cumbersome and inefficient for several reasons. First, most existing programming abstractions for sensor networks are either too low-level (thus too tedious and error-prone) or too high-level (unable to support the diversity of sensor network applications). Second, there is no clear separation between application-level programming and system-level programming. A significant concern is the lack of a general middleware library to isolate application developers from low-level details. Finally, testing sensor network systems is particularly challenging. Sensor systems interact heavily with a (non-repeatable) physical environment, making lab experiments not representative and on-site experiments very costly. This dissertation is targeted for a comprehensive solution that addresses all the above-mentioned problems. The EnviroSuite framework consists of (i) a new programming paradigm that exports environment-based abstractions, (ii) critical middleware services that support the abstractions and separate application programmers from tedious, low-level details, and (iii) testing tools geared for in-situ experimenting, debugging, and troubleshooting.

First, we introduce a new programming paradigm, called *environmentally-immersive programming*

(*EIP*), to capture the common characteristics of sensor network applications, the rich, distributed interactions with the physical environment. EIP refers to an object-based programming model in which individual objects represent physical elements in the external environment. It allows the programmer to think directly in terms of physical objects or events of interest. We provide language primitives for programmers to easily implement their environmental tracking and monitoring applications in EIP. A preprocessor translates such EIP code transparently into a library of support middleware services, central to which are object management algorithms, responsible for maintaining a unique mapping between physical and logical objects.

The major outcome of sensor networks is observations of the instrumented environment, in other words, sensory data. Implementing an application mainly involves encoding how to generate, store, and collect such data. EIP object abstractions provide simple means for programmers to define how observations of the environment should be made via distributed coordination among multiple nodes, thus simplifying data generation. Yet, the next steps, namely, data storage and collection, remain complicated and fastidious. To isolate programmers from such concerns, we also include in the support library a set of data management services, comprising both network protocols and storage systems to allow data to be collected either in real-time or in a delay-tolerant manner.

The final phase in sensor network software development life-cycle is testing, typically performed in-field, where the effects of environmental realities can be studied. However, physical events from the dynamic environment are normally asynchronous and non-repeatable. This lack of repeatability makes the last phase particularly difficult and costly. Hence, it is essential to have the capability to capture and replay sensing events, providing a basis not only for software testing, but also for realistic protocol comparison and parameter tuning. To achieve that, EnviroSuite also provides testing and debugging facilities that enable controllable and repeatable in-field experiments.

Finally, to demonstrate the benefits of our framework, we build multiple representative applications upon EnviroSuite, drawn from both tracking systems such as military surveillance, and monitoring systems such as environmental acoustic monitoring. We install these applications into off-the-shelf hardware platforms and physically deploy the hardware into realistic environments. Empirical results collected from such deployments demonstrate the efficacy of EnviroSuite.

To my parents, Zhangsen and Tianxiang,

and my dear husband, Zhanlu.

Acknowledgments

This dissertation would not have been possible without the help, encouragement, and support of many people.

My first and sincerest gratitude goes to my advisor, Tarek, for his patient and continued guidance, instruction, support, and encouragement throughout the years. He has been such a role model for me – I strive to be as diligent and dedicated for research as he has always been. My special thanks go to Professor Jack Stankovic, for being so supportive throughout and after my graduate studies at University of Virginia. Whenever I turn to him for help, he is always there.

I would also like to thank the other members of my dissertation committee, Professor Klara Nahrstedt and Professor Haiyun Luo, for their invaluable comments and constructive suggestions to improve the work.

I would like to extend my gratitude to Tian He, Chengdu Huang, Qing Cao, Gang Zhou, Lin Gu, and Xue Liu, who contributed a lot to my projects and papers. I would like to especially thank Tian He, Ting Yan, Lin Gu, Radu Stoleru, and Sudha Krishnamurthy for the sleepless nights we have spent together during the demo trips to Florida and Califonia.

Thanks also go to all other members and alumni of the UVa NEST group and the UIUC Cyber Physical Computing group whom I have worked with (in chronological order): Professor Sang Son, Brian Blum, Qiuhua Cao, Qin Chen, Yong Chen, Thao Doan, Raghu Kiran Ganti, Jin Heo, Praveen Jayachandran, Binjia Jiao, Maifi Khan, Ajay Kulhari, Hieu Khac Le, Ying Lin, Ying Lu, Shashi Prabh, Yu-En Tsai, Pascal Vicaire, Anthony Wood, and Ronghua Zhang.

Last but not least, I could not thank enough my parents, who have loved me, fed me, and never left me alone. I could never adequately express my gratitude to my dear husband. The long journey of PhD study has kept us apart for more than half of the years of our marriage (by the way, thank United and American Airlines for the transportation support). I would not have survived it without his endless love and support. This dissertation is dedicated to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This work is to design, implement, and evaluate *EnviroSuite*, an environmentally-immersive programming framework with support for multiple aspects of sensor network software development. We dedicate our efforts not only to programming paradigms that expedite sensor network system design, but to middleware services that isolate programmers from low-level details, and to debugging tools that facilitate in-situ testing and troubleshooting. The ultimate goal is to simplify sensor network programming and reduce software development cost.

In this chapter, we first motivate the work by recognizing difficulties in sensor network programming (Section 1.1). We then give an overview of a comprehensive solution to aid sensor network programming from multiple aspects (Section 1.2, 1.3, 1.4). Next, we highlight the major contributions of this dissertation (Section 1.5). Finally, we present an architectural view of the contributions as well as outline the rest of the dissertation (Section 1.6).

1.1 Motivation

Recent advances in low-power processing units, low-power communication, and sensor technologies have revealed a rich design space of networked, embedded sensors. Such sensing devices will be able to monitor a wide variety of physical conditions: light, temperature, humidity, acceleration, motion, noise, etc. In addition to sensing, such devices can also be equipped with processing, wire/wireless communication, and storage capabilities. Deeply embedded into the environment, such devices are able to represent the physical world in bits, seamlessly integrating the physical world into the cyber world. Furthermore, the increased miniaturization and reduced cost of hardware (by Moore's Law) make it viable to use such devices to instrument the environment at unprecedented granularities and scales. As a result, in recent years the field of sensor networks keeps expanding and thriving, attracting tremendous interests from both the research community and industry.

A major challenge in sensor network research is the extreme difficulties in application development. Sensor networks lie in the intersection of multiple territories: distributed computing, wireless networks, embedded and ubiquitous computing, and real-time systems. Programming of sensor networks combines the difficulties of these existing disciplines, and is thus more challenging than any of them individually. We enumerate the major factors that complicate development of sensor network systems as follows.

- Event-driven programming

  Miniaturized sensing devices usually subject to severe constraints in power, CPU bandwidth, and memory. At the same time, such devices have to handle high levels of concurrency originated from radio communication and high-frequency sensory sampling. Traditional ways of handling concurrency via multiple threads give rise to large memory footprint and heavy CPU load, thus not appropriate for sensor networks. Instead, event-driven approaches [70] are more common for systems with severe memory constraints, since they do not require that memory space be reserved for each execution thread. For this reason, many operating systems for sensor networks [40, 19, 35] adopt event-based models to handle high levels of I/O concurrency. Such event-based models can not support blocking or spin loops as multi-threaded model does. Alternatively, many operations have to be done in a phase-splitting way: a function call that requests certain action returns immediately, and an interrupt comes back later to signal completion of the requested action. In such models, programmers have to frequently use explicit state machines to manage control flow, making programming extremely difficult [53, 90, 20].

- Low reliability

  Sensor network applications, exposed to unpredicted and dynamic operating environments, must handle a wide variety of failures [34]. For instance, hardware failures can partially or completely disable a node. Network dynamics caused by changing of the surrounding environment or node mobility may lead to busty traffic, link losses, and potentially network partitions. System reconfiguration such as adding or removing nodes can also alter operation state of the network. It is a nontrivial task to program a sensor network application to be resilient to all the possible failures.

- Collaborative operation

The low cost and miniaturized size of sensing devices determine that the resources available to individual nodes are severely limited. The high-resolution deployment of sensor networks implies that each node has only very limited sensing capabilities in terms of accuracy and range. Therefore, executing complex tasks, such as tracking a vehicle, often requires collaboration among multiple nodes. It is notoriously difficult to decompose the desired global behavior (e.g., tracking a vehicle) into local behaviors of individual nodes and their collaborations [33]. Note that the alternative centralized solutions, where each node serves only as dumb data collectors and complex processing of the collected data is done at a centralized point, are not as scalable and usually waste large amount of energy in data collection. They are thus not appropriate for most sensor network applications with severe energy constraints [22].

- Limited runtime visibility

  Another issue of concern confronting application programmers for sensor networks is the lack of large display monitors. Whereas most of the off-the-shield monitors for personal computers or servers measure more than 20 inches in current days, typical sensing devices (e.g., Mica motes [15], Tmote [72], and Intel Mote [45]) have very tiny display screens or even no screens. As a result, there is very limited or no runtime visibility. Previously simple programming routines as monitoring or debugging a running program becomes extremely hard.

- Non-repeatable environmental input

  Sensor networks operate on and interact with dynamic and asynchronous events in the physical environment, which are hard to reproduce exactly. Repeatability of such environmental events is essential for debugging or troubleshooting of sensor network systems, especially for event-driven systems where the state of execution changes depending on the particular sequence of environmental events received and their timing. Matters as tuning protocol parameters for particular event scenarios or comparing performance of different protocols require controlled, repeatable experiments. However, in the face of environment dynamics, it is a nontrivial proposition to achieve that.

These factors, as a whole, make sensor network software development costly, error-prone, and time-consuming. As a result, current sensor network developers are limited to computer science experts with intimate knowledge of sensor networks and excellent system skills. To open sensor network techniques to the masses, we have to provide enough programming facilities. Hence, the goal of the dissertation is to design and implement a new programming environment which (i) raises the level of programming abstractions to be more intuitive and less error-prone compared with event-driven programming, (ii) comprises a library of middleware services to support the abstractions and to hide the details of failure handling and collaborative computing from application programmers, and (iii) provides debugging facilities that resolve the issues of non-repeatable input and limited runtime visibility. We call the new programming framework, *EnviroSuite*.

1.2 Programming Paradigms

Traditional programming languages such as Java and C, as well as their sensor network adaptations, such as nesC [27] and its extension galsC [12, 13], are too low-level. While allowing for more flexibility, they tend to expose too many details to application programmers and entail large overhead in application design and implementation. Their basic computation, communication and actuation unit is typically the sensor node. Programmers must think in terms of single node activities and explicitly encode interactions among nodes. For example, programmers are exposed to reading sensing data from appropriate sensor devices, aggregating data pertaining to the same external stimulus, deciding where to send it, and communicating with actuators if needed. If the monitored activity moves in the environment, programmers are responsible for spatial and temporal correlation of measurements obtained about the activity across a changing set of sensor nodes, and associating that data with event progress.

A more desirable approach would be for the programmers to encode only overall network behavior, leaving it to the underlying system to decompose such behavior into node-level algorithms. Examples of higher-level abstractions that address this concern include logical-node-based primitives [33], token-based languages [73], database-centric abstractions [66, 99, 10], event-based systems [55], group-based primitives [9, 95, 92, 58], state-centric approaches [56] and virtual machines [51, 11]. These paradigms offer logical nodes, tokens, queries, events, sensor node groups, logical state, and virtual machines, respectively, as the underlying abstractions with which the programmer operates. However, most paradigms failed to capture the unique properties of sensor networks: their distributed interactions with the physical environment.

One of the goals of the dissertation is to develop a new programming paradigm that exposes abstractions revolving directly around physical elements of the environment, as opposed to sensor network constructs such as regions, neighborhoods, or sensor groups.

Therefore, we propose and implement one of the first high-level programming paradigms that combine elements in the physical world tightly with programmable objects in the logical world. We call it *environmentally immersive programming* (*EIP*) [61, 60]. EIP is an object-based programming paradigm. Its object abstractions revolve directly around elements of the physical environment. In other words, each EIP object represents certain element in the physical world. Dynamic object instances are created automatically by the run-time system when the corresponding external elements are detected, and are destroyed when these elements leave the network. A one-to-one mapping between object instances and the corresponding environmental elements is maintained by the system. Object instances float across the network following (geographically) the elements they represent. Objects encapsulate the aggregate state of the elements they represent, making such state available to their methods. Object methods are executed at the location of the corresponding physical element, which is ideal for sensing and actuation tasks. These objects (as opposed to the individual nodes) are therefore the units that encapsulate program data, computation, communication, sensing and actuation. The existence of the sensor network is thus made more transparent. By providing the object abstractions, EIP successfully isolates application programmers from tedious, node-level details.

1.3 Support Middleware

To support EIP abstractions, we provide a library of middleware services. Apparently, the most critical set of services is object management algorithms that maintain the mapping between environmental elements and objects. They are also responsible for obtaining aggregate state and executing object code. However, these services are not sufficient enough.

Data is the major outcome of sensor networks. Sensor network systems are centered around how to generate, store, and collect observations of the physical world. Using EIP object abstractions and the underlying management algorithms, programmers can easily write code to define how such observations should be generated as aggregate states of interested physical elements. The next challenge is to collect such states so as to present them to operators of the network. Therefore, another critical set of services should be data collection services to isolate programmers from such concerns. The data management services we investigate include not only multi-hop communication services to communicate data back in real-time to back-end data monitoring centers (i.e., *base stations*), but also data storage services operating in a store-forward manner for delay-tolerant [23, 46] applications.

Data collection has been addressed at length in sensor network literature. The assumption has traditionally been that a sensor network is connected to a base station that collects the data. This assumption is suitable when near-real-time information availability is desirable. Tracking and event notification applications, for example, fall under this category. A significant number of applications, however, do not require real-time information [59, 88, 25]. For example, an environmental scientist interested in studying light variations on the forest floor due to canopy closure in the Spring might deploy a sensor net and collect the data only months later when the experiment is over[1]. This "fisherman's net" model of the application allows for significant architectural simplifications of the monitoring infrastructure. Most importantly, there is no longer a need to maintain a base station in the field. We call the above, a *disconnected network model*.

Comparing with the connected network model, this disconnected model imposes a different set of challenges. Rather than focusing on multi-hop communication protocols to carry data back to base stations, in such model, programmers are more concerned with how to leverage in-network storage (already available on sensor nodes as flash memory) and how to opportunistically upload data if opportunities (e.g., approaching of data mules) arise. Therefore, besides providing multi-hop communication services for connected operation model for the purpose of completeness, another major goal of the dissertation is to develop data storage and collection services that address the unique concerns of the disconnected network model.

In this dissertation, we present *EnviroStore* [63], a cooperative storage service for sensor network applications geared for disconnected operation. The primary concern of the service is to maximize effective storage capacity of the network while disconnected. EnviroStore takes into account various factors that may cause data loss. First, it employs data redistribution schemes to optimize sharing of network storage. Observe that some nodes will record more data than others. This may be due to asymmetry in environmental inputs (e.g., acoustic nodes near sound sources will fill up before those in quiet areas), or due to data-dependent variations in compression ratio of algorithms such as run-length encoding. Data loss can be minimized by migrating data from nodes that are full to those that are not, as well as by exploiting upload opportunities when available. Second, EnviroStore also takes into account the rate of energy consumption to avoid depletion-related data loss. Naturally, nodes will be unable to store data after energy is depleted.

[1]This is an actual study that has been performed in Trelease Woods near UIUC campus in Spring 2006

EnviroStore further reduces the burden of EIP programmers. Via EnviroStore, the aggregate state of the environmental elements encapsulated in objects or the execution results of object methods can be easily deposited in the network and collected automatically later. This is especially useful for EIP programmers to develop applications where direct connection to base stations is not available or rather intermittent. It has to be noted that though EnviroStore is a critical component of the EnviroSuite framework, we do not preclude the use of EnviroStore in other programming models.

1.4 Debugging Facilities

With sufficient programming support in place, rapid design and implementation are made possible by simply utilizing the high-level and human-intuitive abstractions and underlying middleware services. Yet, the development process is not complete without debugging your system thoroughly. Sensor network systems are often embedded and expected to run continuously for months or years without fatal failures. Therefore they have to be tested more carefully than those for personal computers or servers (which are closely attended by home users or system administrators). However, testing sensor systems is much harder than testing PC software. As mentioned before, one of the major reasons is the non-repeatability of environmental inputs. Quite different from controlled lab settings, physical environments introduce much higher degree of uncertainty and much higher probability of failures. The limited runtime visibility of sensor network applications further exacerbates the situation. As the number of applications developed and deployed keeps growing, there is an increasing need for tools and services to assist with testing and troubleshooting of applications in targeted deployment environments. Developing such tools is therefore the third goal of the dissertation.

To facilitate the debugging of event-driven systems, we design and implement a distributed service that improves repeatability of experimental testing of sensor networks, called *EnviroLog* [62]. EnviroLog can capture and reproduce distributed events on demand. It provides the abstraction of a completely repeatable environment as observed by the sensory subsystem for the sake of experimental testing. Communication properties remain stochastic. Hence, a separation is achieved between the effects of communication non-determinism and the effects of environmental non-repeatability in the study of sensor network protocols. The service is especially valuable in the study of rare, unsafe, or hard-to-reproduce events such as the motion of tracked animals through a sensor field.

Recall that objects in EIP are logical representations of external environmental events. The repeatability of such events (made possible by EnviroLog) is particularly useful in evaluating both the EIP support library itself and applications built upon the EIP model.

1.5 Contributions

This dissertation presents our attempt to build a practical development framework centering around environmentally-immersive programming, and demonstrates the efficacy of the work by several integrated systems that are built upon EnviroSuite and are actually deployed into the physical environment. Specifically, we advance the State-of-the-Art in the following ways:

- **Environmentally-immersive programming paradigm:** Different from most of the existing ones, our newly proposed paradigm captures the unique characteristics of sensor networks, their rich, distributed interaction with the physical environment. EIP programmers directly operate on environmental elements rather than network constructs. In the history of programming languages, designers have been struggling to raise abstraction levels from machine languages, to assembly languages, and to high-level languages such as C++ and Java. Our paradigm further pushes programming abstractions closer to human brains and application contexts in terms of sensor network programming.

  In a broader context, the paradigm is applicable to not only sensor network systems but other event-based systems whose main functionality is to respond to certain measured conditions. Heo et al. [38] shows a recent effort to apply a similar idea of environmentally-immersive programming to build a server management system to optimize energy consumption in server farms.

- **Distributed in-network storage solution for disconnected operation:** The prevalence of sensor network applications adopting a disconnected operation model advocates a paradigm shift from traditional communication-centric sensor networks to storage-centric sensor networks. In-network storage solutions are essential in storage-centric sensor networks. We present EnviroStore, a storage service optimized for disconnected operations. It enables EIP programmers to easily deposit aggregate states of objects in the network for future retrieval. The service is interoperable with a wide range of sensor network applications via its generic interface.

- **In-situ testing support by achieving repeatability of asynchronous events:** Sensor network systems usually operate on asynchronous events originated from environmental dynamics, making debugging and testing of such systems extremely challenging. EnviroLog achieves the repeatability of the dynamic, asynchronous events by capturing and replaying them on demand. With EnviroLog, transient failure scenarios resulting from specific environmental conditions become repeatable and tractable. EnviroLog also greatly simplifies in-field experiments by removing the need to manually regenerate physical events (such as driving a car through the field hundreds of times).

- **Evaluation of proposed solutions through deployed practical systems:** One important objective of the dissertation work is to provide usable sensor network development support for the community. We thoroughly validate the usability and practicability of the proposed methodologies through multiple system scenarios. The techniques presented are applied to build both a large-scale surveillance system and an acoustic monitoring application. These systems are uploaded to actual hardware platforms and then deployed into realistic physical environments for demonstration and evaluation purposes. The empirical results from the deployments demonstrate the efficacy of the EnviroSuite framework itself as well as its capabilities to support the diverse set of sensor network applications.

1.6 Dissertation Outline

Figure 1.1 provides an architectural view of EnviroSuite. The EnviroSuite framework sits on top of sensor hardware platforms and operating systems, comprising three major components: (i) the *Environmentally-Immersive Programming* paradigm that supports an object-based programming model, (ii) a set of *support middleware* services that isolate various low-level concerns, and (iii) a set of *debugging tools* that facilitate debugging, tuning, and evaluating of real systems. In addition to that, we also provide the *EnviroSuite preprocessor* that translates EIP code into native application code (e.g., nesC code) with debugging add-ons by configuring and restructuring algorithms from the support library of middleware services.

The EIP paradigm component mainly comprises language primitives for EIP object abstractions and the *EIP support library* that mainly contains the variety of object management algorithms to support the abstractions at runtime. Inside the support middleware library are a multi-hop communication service, called *Multiple-Source Transport Service* (*MSTP*) [39], for real-time applications to communicate data during runtime, and a storage service, called *EnviroStore*, for delay-tolerant applications to store data in the network and retrieve at a later point. In terms of debugging facilities, we include two tools in the EnviroSuite frame-

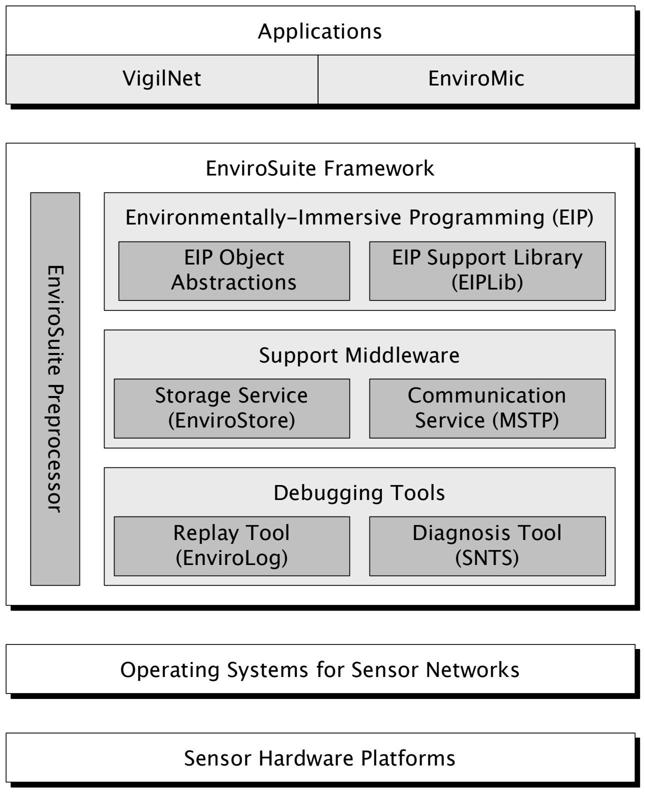

Applications
VigilNet
EnviroMic
EnviroSuite Framework
EnviroSuite Preprocessor
Environmentally-Immersive Programming (EIP)
EIP Object Abstractions
EIP Support Library (EIPLib)
Support Middleware
Storage Service (EnviroStore)
Communication Service (MSTP)
Debugging Tools
Replay Tool (EnviroLog)
Diagnosis Tool (SNTS)
Operating Systems for Sensor Networks
Sensor Hardware Platforms

Figure 1.1: System Architecture of EnviroSuite

work: one is a replay tool called *EnviroLog*, and the other is a failure diagnosis and troubleshooting tool based on machine learning techniques, called *SNTS* [50]. EnviroLog can capture and replay asynchronous environmental events on demand, thus providing repeatable inputs for testing and evaluation purposes. SNTS is a tool that performs automated failure diagnosis for sensor network systems. Note that MSTP and SNTS are collaborative work mainly contributed by other colleagues. We do not claim them as major contributions of this dissertation. Details on the two services can be found in [39, 50].

The EnviroSuite framework is utilized in building several integrated systems, including *VigilNet*, a military surveillance system, and *EnviroMic*, an acoustic monitoring application.

The rest of the dissertation is organized as follows. Chapter 2 elaborates the environmental immersive programming abstractions and the support library of distributed object management algorithms. Chapter 3 describes EnviroStore and Chapter 4 presents EnviroLog. Chapter 5 discusses several integrated systems built upon EnviroSuite and presents evaluation results collected through field experiments. Chapter 6 supplies an extensive literature survey. Chapter 7 concludes the dissertation.

Chapter 2

Environmentally-Immersive Programming Paradigm

2.1 Introduction

The need to facilitate software development for sensor networks motivates new high-level abstractions for programming-in-the-large. These abstractions must hide the details of distributed algorithms that observe and track physical elements, capture the unique properties of sensor networks such as their distributed interactions with a physical environment, and address the issue of scale.

Current sensor network programming practice remains very cumbersome and inefficient using low-level languages such as nesC [27]. They usually require programmers to explicitly encode the behavior of individual nodes and the interactions among them. Apparently, programming in such node-based languages is too low-level, too error-prone, and results in long programs that expose too many tedious details to application developers. To make development easier, a multitude of high-level programming paradigms and abstractions have been proposed, including logical-node-based [33], token-based [73], database-centric [66, 99], event-based [55], group-based [95, 92], and virtual-machine-based [51] approaches. However, most of them are inadequate to support the diversity of sensor network applications. Some are primarily inspired by one application model, thus inapplicable to other application scenarios. Some are not as intuitive since they fail to capture the unique characteristics of sensor network applications. Certainly, there is a need to find new solutions that expose the right level of abstractions without sacrificing the flexibility to support a broad set of application models.

Therefore, in this chapter, we propose a new programming paradigm called *Environmentally-Immersive Programming* (EIP), whose abstractions revolve directly around elements of the environment as opposed to sensor network constructs such as regions, neighborhoods, or sensor groups. EIP is an object-based programming paradigm. However, it differs from other object-based systems in that its objects are representations of elements in the external environment. The creation and deletion of object instances are correspondent to the appearance and disappearance of external elements in the sensor field. The runtime system of EIP maintains a unique mapping between object instances and the environmental elements despite the movement of the elements. Classical objects (that do not represent any environmental elements) are also supported.

EIP abstractions are supported by an underlying library of distributed algorithms, which are implemented in nesC on TinyOS [40], an operating system designed specifically for sensor networks. We evaluate the support library as well as several small applications written in EIP both on TOSSIM [52] and on a mote-based sensor network. TOSSIM is an emulator that runs the actual nesC service code, emulating on a PC the behavior of programs on the Mica motes [15].

The remainder of the chapter is organized as follows. Section 2.2 gives an overview of the EIP model. Section 2.3 provides a detailed description of the exported abstractions. Section 2.4 presents the design of the essential algorithms underlying these abstractions. Section 2.5 highlights some of the implementation details. Section 2.6 presents and analyzes performance evaluation results. Section 2.8 concludes the chapter.

2.2 Overview of EIP

EIP lets the programmer think in terms of objects in the external environment that are relevant to the application. An environmental object may refer to a localized entity (such as a moving vehicle) or a distributed region of the environment (e.g., a geographic area or an area specified by some sensory signature such as high temperature). Typically, the system must keep some state or other information about each object. This state is maintained in variables encapsulated within the object that can be accessed using object methods. Hence, each object is given by (i) a sensory or geographic signature that defines its boundaries or location, (ii) a set of data variables to be collected in its vicinity, and (iii) a set of methods that can be performed in its context. The fact that the obtainment of values stored in the encapsulated variables and the execution of local methods may need distributed computation (such as data aggregation) is hidden from the programmer. The programmer may also define regular objects that are not linked to objects in the environment. We call them *function objects*. Environmental objects and function objects seamlessly coexist in the programmers' world and can invoke each other's methods using a remote object invocation mechanism. An example of such mapping is depicted in Figure 2.1.

The example in Figure 2.1 represents a surveillance application that monitors vehicle and person move-

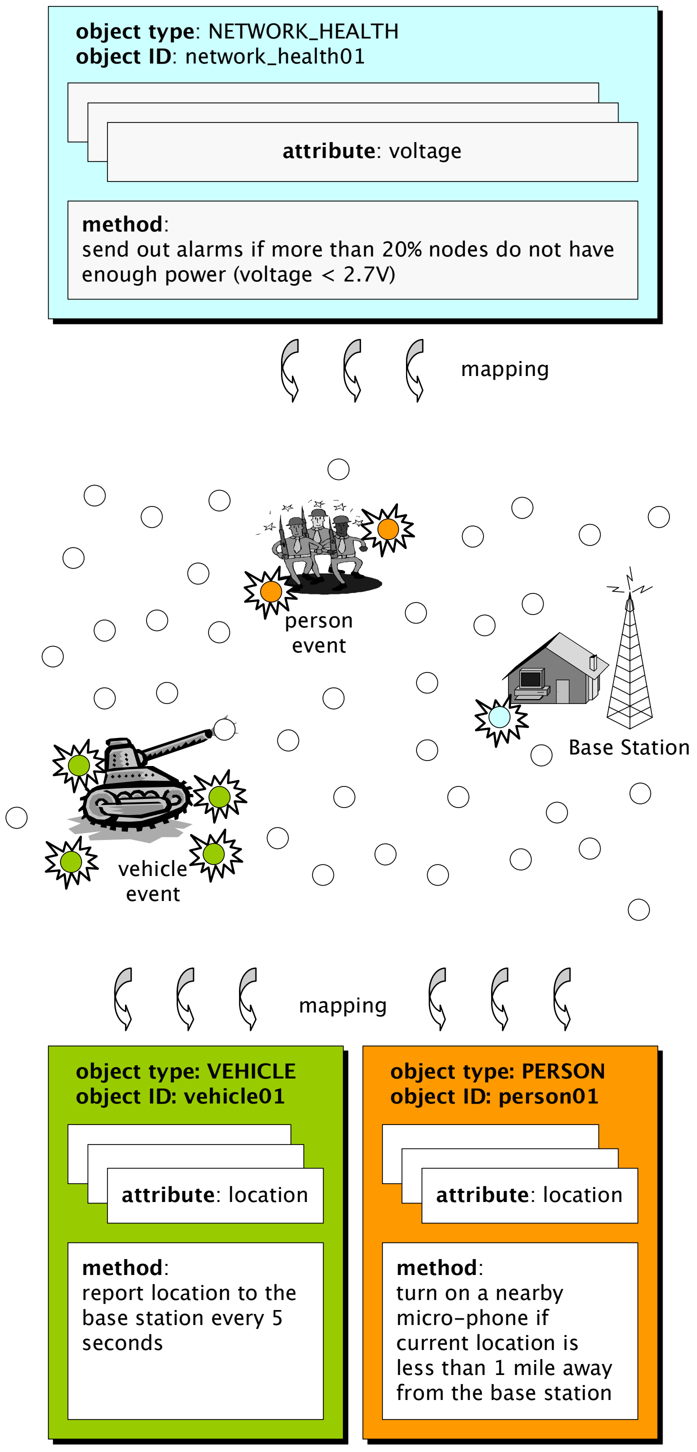

object type: NETWORK_HEALTH
object ID: network_health01
attribute: voltage
method:
send out alarms if more than 20% nodes do not have enough power (voltage < 2.7V)
mapping
person event
Base Station
vehicle event
mapping
object type: VEHICLE
object ID: vehicle01
attribute: location
method:
report location to the base station every 5 seconds
object type: PERSON
object ID: person01
attribute: location
method:
turn on a nearby micro-phone if current location is less than 1 mile away from the base station

Figure 2.1: One-to-one mapping between physical events and event objects

ment in a hostile territory (e.g., behind enemy lines). Each tracked vehicle or person is mapped into a dynamically instantiated object with a unique label, denoted by an object ID in EIP. Desired event attributes such as location can be returned for the object. This application also periodically monitors the health of the network by collecting information on nodes that are alive and their remaining power. The network is thus mapped into an object that maintains aggregate health statistics. Computation, communication and actuation can be logically attached to these objects. Examples include reporting vehicle location by vehicle objects, turning on microphones in their vicinity for tracking purposes, or sending out alarms if system health fails to meet an acceptable threshold.

These primitives are supported by the EIP support library (*EIPLib*), which contains the detailed implementations of primitive algorithms (such as sensor data processing and aggregation) and some other high-level services (such as object maintenance and inter-object communication). A preprocessor (*EIP preprocessor*) is introduced to translate EIP code into nesC. The relation among EIP, EIPLib and the EIP preprocessor is depicted in Figure 2.2.

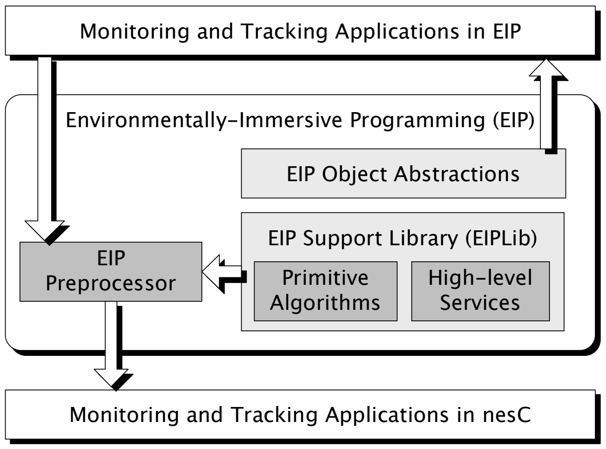

Monitoring and Tracking Applications in EIP
Environmentally-Immersive Programming (EIP)
EIP Object Abstractions
EIP Support Library (EIPLib)
EIP Preprocessor
Primitive Algorithms
High-level Services
Monitoring and Tracking Applications in nesC

Figure 2.2: Relation of EIP, EIPLib & EIP preprocessor

Programmers design and implement environmental monitoring and tracking applications using a combination of EIP and nesC. Taking such implementations as input, the EIP preprocessor configures and restructures services and protocols in EIPLib to automatically produce as output the corresponding implementations in the nesC language. The resulting code can be compiled on TinyOS and uploaded to the motes. In the following sections, we describe in more details the abstractions of EIP, the services and protocols provided in EIPLib to support these abstractions, and the language translation carried out by the preprocessor.

2.3 EIP Syntax

When a programmer develops a monitoring application using EIP, the programmer creates a virtual world with a set of logical objects, which attempts to reflect the real world with a set of physical objects. Each object is defined by a sensory signature such that contiguous groups of nodes that satisfy that signature will be given a unique object ID. These object IDs constitute a global name space available to the programmer. Various data operations can then be performed on different locales defined by the corresponding object IDs. These operations are typically coded as methods encapsulated in the corresponding objects. Observe that EIP is only concerned with (i) grouping together nodes that satisfy programmer-defined sensory signatures, (ii) giving global names to those groups, (iii) executing programmer-defined data operations within each named group, and (iv) storing programmer-defined group state in variables encapsulated within the named objects. The correspondence between the groups and meaningful aspects of the environment is the programmer's responsibility. For example, there is inherent uncertainty regarding whether or not a motion and magnetic signature defined by the programmer truly signals the presence of a vehicle. EIP is concerned only with tracking the defined signature. The uncertainty in the interpretation of the signature must be handled at a higher-level.

Syntactically, an EIP program consists of a list of object declarations such as the one shown in Figure 2.3. Each declaration specifies a user-defined object *type*, an object *condition* statement, declaration of object *attributes*, and the object *methods*.

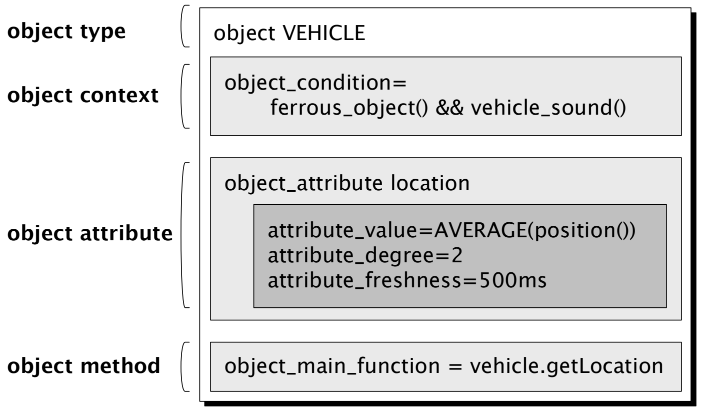

object type
object context
object attribute
object method
object VEHICLE
object_condition=
ferrous_object() && vehicle_sound()
object_attribute location
attribute_value=AVERAGE(position())
attribute_degree=2
attribute_freshness=500ms
object_main_function = vehicle.getLocation

Figure 2.3: Object declaration of object VEHICLE

The object condition statement creates a mapping between the software object and the corresponding environmental element. For example, it can specify the sensory signature of an external tracked entity, or

Object Context	`object_condition`

Object Attributes	`object_attribute`
	`attribute_value`
	`attribute_degree`
	`attribute_freshness`

Object Methods	`object_main_function` `object_function`

Table 2.1: Keywords for basic EIP object declaration

a geographic area defining a physical region. An object is created for each contiguous region where the object condition is true. A contiguous region is one that is not partitioned. In other words, there exists a path between any two nodes in the region that has no intermediate hops outside the region. We call this region the object *context*. A null object condition specifies that this object is not a representation of an environmental element (e.g., a pure computational object), which is called a function object, as mentioned above.

Specifications of object contexts are followed by declarations of encapsulated data to store the state of the object. Such data typically refers to aggregates of sensory measurements or node attributes over the object context. They can be thought of as query results over the context. The declaration specifies the method of aggregation together with confidence and freshness parameters. Finally, as in traditional object-based systems, an object *main* function specifies the name of a default constructor method to be automatically executed when the object is created. Other methods can be defined to be executed when called. Object methods can access the attributes of their encapsulating object and perform remote method invocations on other objects.

Objects are instantiated either statically or dynamically. The former is useful to represent fixed environmental elements such as topological features of the terrain. The latter is useful, for example, to represent dynamically arriving targets in the environment. As described later in this chapter, special care is taken to ensure unique representation (i.e., that a single object is instantiated to refer to a single target, even though the target causes multiple sensor hits).

EIP keywords for basic object declarations are listed in table 2.1. More detailed discussion on EIP object contexts, attributes, and methods is presented in the following subsections respectively, using the object declaration example depicted in Figure 2.3.

2.3.1 Defining the Object Context

In EIP, the object condition statement defines the object context, which is the continuous region where the object condition is true. The EnviroSuite framework includes a library of sensor data processing algorithms (called the *condition library*) designed by domain experts for purposes of defining object contexts. These algorithms return (possibly) filtered or otherwise processed sensor outputs (e.g., `temperature()`), or identify specific boolean environmental conditions (e.g., `ferrous_object()` or `vehicle_sound()`), or return node attributes (e.g., `position()` or `voltage()`). A boolean expression of such conditions can then define the region of object context. We call it the object condition statement. For example, the following declaration defines the condition that represents the potential presence of a vehicle:

```
object_condition = ferrous_object() && vehicle_sound();
```

In this example, `ferrous_object()` is a function that returns true when the magnetometer output indicates a significant disturbance to the earth magnetic field (consistent with the passage of a large ferrous object), and `vehicle_sound()` indicates microphone output of energy and pitch consistent with the sound of a passing vehicle. Implementation of such functions is described in [31].

The idea is to compile a library of such conditions to abstract the specifics of sensor processing away from the programmer in much the same way device drivers abstract the details of device I/O away from application code. The separation between high-level application code and low-level sensor processing comes at the cost of increased condition library size, since many different algorithms need to be written to identify a sufficient range of useful conditions for each sensor type. This is not unlike the proliferation of device drivers (one for each version of every possible device) in contemporary operating system installations. The success of device drivers as a means for separating concerns leads one to believe that the condition library will considerably simplify application development in sensor networks. An object executes when and where the conditions defined in its condition statement become true.

Observe that conditions can also be parameterized. For example, the condition:

```
object_condition = altitude() > 500 && temperature() < 32;
```

defines the region (i.e., object context) that satisfies freezing temperatures on top of a local hill. The case of `object_condition = NULL` specifies a function object not associated with an environmental element (region or physical entity).

2.3.2 Defining Object Attributes

The main purpose of objects invoked in response to environmental conditions such as those mentioned above is usually to monitor attributes of environmental events, targets or regions. These attributes are measurements collected and aggregated by nodes in the object context. Specification of attributes requires specification of (i) the sensor measurements in question, and (ii) optionally, their method of aggregation. Aggregation is always performed over all nodes within the object context. The sensor measurements to be aggregated could be any environmental measurements, or node attribute measurements such as remaining battery power or node position, for which a measurement function exists in the condition library described above. A library, called the *aggregation method library*, is supplied, which lists a set of aggregation methods such as `AVERAGE`, `MAX` and `RANGE` on attributes. For example, to define an aggregate attribute, `targetLocation`, EIP programmers can simply specify the corresponding node measurement, `position()`, from the condition library, and the name of the appropriate aggregation method, say `AVERAGE`, from the aggregation method library, in an object attribute clause, such as:

```
object_attribute targetLocation {
    attribute_value = AVERAGE(position());
}
```

Within the declaration of an attribute, EIP allows the programmer to specify the minimum aggregation degree, `attribute_degree`. The aggregate attribute is valid only when it is the aggregation result from at least as many nodes as `attribute_degree`. This knob allows programmers to control the confidence in retrieved information. The feature is especially useful in reducing false alarms. Another important property of attributes is freshness. Most monitoring applications have temporal data validity constraints. Usually, stale information is of no use. EIP allows programmers to define `attribute_freshness`, which determines how often aggregate attributes are to be sampled and updated by the mechanisms that compute them in EIPLib.

2.3.3 Defining Object Methods

Sensor network applications can have more complex functionality than merely monitoring attributes. In general, computation, communication or actuation could be encapsulated into the definition of an object.

EIP tries to make full use of existing general-purpose languages, such as nesC, and their existing modules, such as those exported by TinyOS, by separating real object method implementation from object declaration. In object declaration, EIP programmers are required to denote the name of functions implementing in a general language the object methods. Such functions can use the EIP communication primitives (using keyword `ES_IOC` and `ES_IOCRESULT`) and read values of encapsulated aggregate attributes of the object (using keyword `ES_GETATTRIBUTE`). The separation of object method declaration and object method implementation retain independence of EIP abstractions from the underlying programming language.

There are two types of object methods that can be encapsulated within an object. Those object methods specified in `object_main_function` statements are functions which will be automatically executed upon the creation of the corresponding object. In contrast, object methods specified in `object_function` statements will be executed only when they are called by other objects. Assuming programmers choose nesC as the general language to implement object methods, the following clause specifies that the implementation of the main object method can be found in nesC command `getLocation` within interface `vehicle`.

```
object_main_function = vehicle.getLocation;
```

To facilitate communication and coordination beyond the scope of one object, we introduce a RPC-like mechanism in EIP, called the *Inter-Object Call* (IOC). IOC is different from traditional RPC in several aspects. First, both the caller and the callee of IOC can be migrating across nodes as the location of the external object changes. Such migration is transparent to programmers, who simply specify the callees instance name (to be stated below) and never worry about which physical nodes these objects are located on. Second, IOC is asynchronous. Callers do not block themselves to wait for results. Instead, results declare their arrivals by interrupts. The keyword for IOC is `ES_IOC` and `ES_IOCRESULT`. The former is used for executing an IOC and declaring its handler and the latter for receiving IOC result interrupts. All object methods defined in an object can be remotely called by any other objects by using its reference. The underlying low-level communication protocol and routing extensions to support IOC can be found in [9].

2.3.4 Defining Static Object Instances and Global Variables

The above discussion covered declaration of object types. Objects that represent fixed environmental elements, such as topological features of the terrain, can be statically instantiated. These static instances can be used, for example, as the destinations of IOCs that invoke object methods. Object types that do not have static instances will be instantiated dynamically at run-time when their object conditions become true. They would have to send their handle to any other objects that need to communicate with them.

EIP also allows programmers to define globally shared static variables in (static) object declarations and to access defined static variables in object method implementation by using EIP keyword, `ES_READ` and `ES_WRITE`.

The next section gives a complete tracking application implemented in EIP, including code samples for static instances and static variables.

2.3.5 A Tracking and Monitoring Application in EIP

A typical tracking and monitoring application written in EIP (and some nesC) is shown in Figure 2.4. The main function of this application is to estimate the current location of a tracked vehicle, update the estimates every 500 ms and report the estimated location to the base station every 500 ms. The total number of vehicles is counted. At the same time, voltage values for individual nodes are collected every 20 minutes to obtain system health information. This application illustrates the main abstractions supported by the framework, as well as the programming style.

The application declares three object types `VEHICLE`, `NETWORK_HEALTH` and `MONITOR` which refer to a dynamically instantiated object, a geographic region object and a function object, respectively (lines 1 - 20).

For object type `VEHICLE`, the `object_condition` statement (line 2) specifies its sensory signature as `ferrous_object()` and `vehicle_sound()`. The `object_attribute` statements (lines 3 - 6) define an aggregate attribute `location` for which the value is the average of positions of more at least 2 nodes, updated every 500 ms. The `object_main_function` statement (line 7) states that main object method implementation can be found in interface `vehicle` that includes command `getLocation`.

For object type `NETWORK_HEALTH`, the `object_condition` statement (line 9) specifies its object context as `TRUE` to include all nodes in the network. The `object_attribute` statements (lines 10 - 12) define an attribute `energyLevel` as the voltage values of individual nodes with an update rate 20 minutes. The `object_main_function` statement (line 13) defines that main object method is command `getEnergyLevel` in interface `networkHealth`, which obtains an array of node IDs and voltage values. Line 14 creates a static instance `networkHealthInstance` for object type `NETWORK_HEALTH` so

Object Declarations

```
object VEHICLE {
    object_condition = ferrous_object()&&vehicle_sound();
    object_attribute location {
        attribute_value = AVERAGE(position());
        attribute_degree = 2;
        attribute_freshness = 500ms; }
    object_main_function = vehicle.getLocation; }
```

```
object NETWORK_HEALTH {
    object_condition = TRUE;
    object_attribute energyLevel {
        attribute_value = voltage();
        attribute_freshness = 20m; }
    object_main_function = networkHealth.getEnergyLevel; }
static NETWORK_HEALTH networkHealthInstance;
```

```
object MONITOR {
   object_condition = NULL;
   object_main_function = monitor.start;
   object_function = monitor.reportLocation;
   static int vehicleNumber = 0; }
static MONITOR monitorInstance;
```

Implementations of Object Methods

object method implementation of object VEHICLE

```
Triple_float_t *currentLocation;

command result_t vehicle.getLocation() {
    call ES_WRITE(monitorInstance.vehicleNumber,
                           monitorInstance.vehicleNumber +1);
    return call Timer.start(TIMER_REPEAT, 500); }

event result_t Timer.fired() {
    currentLocation = call ES_GETATTRIBUTE(location);
    ES_IOC report = call monitorInstance.monitor.
                               reportLocation(currentLocation);
    return SUCCESS; }

ES_IOCRESULT report(bool result) {
    //deal with remote call results here
    return; }
```

object method implementation of object NETWORK_HEALTH

```
uint16_t currentEnergyLevels[MAX_NODE_NUMBER];

command result_t networkHealth.getEnergyLevel() {
    return call Timer.start(TIMER_REPEAT, 1200000); }

event result_t Timer.fired() {
    currentEnergyLevels = call ES_ATTRIBUTE(energyLevel);
    //deal with obtained node IDs and voltage values here
    return SUCCESS; }
```

object method implementation of object MONITOR

```
command result_t monitor.start() {
    return SUCCESS;  }

command bool monitor.reportLocation(Triple_float_t
Location) {
    //deal with received target location here
    return TRUE; }
```

Figure 2.4: An EIP application

that it will be instantiated statically in system initialization and IOCs can be made through this reference.

For object type `MONITOR`, the `object_condition` statement (line 16) specifies `NULL` as the object context since the object is not mapped to any environmental element. The `object_main_function` statement (line 17) specifies the command `start` in interface `monitor` as the main object method. Finally, the `object_function` statement (line 18) defines that command `reportLocation` in interface `monitor` can be remotely called by any other objects by using IOC and its static instance `monitorInstance` (line 20). Line 19 defines a static variable `vehicleNumber` which is globally accessible by any object through `ES_READ` and `ES_WRITE`.

In the object method implementation of object `VEHICLE`, it is defined that static variable `vehicleNumber` is increased by one whenever a new instance of `VEHICLE` is created (line 23). For each instance, every 500 ms (line 24) the current value of aggregate attribute `location` is fetched (line 26) and sent to the base station by using `ES_IOC` (line 27) to remotely call method `monitor.reportLocation` located in static instance `monitorInstance`. In line 29, `ES_IOCRESULT` keyword is used to receive IOC interrupts of `ES_IOC report`. The interrupt handler name must be the same as `ES_IOC` which is `report` and the parameters should be of the same type as the returned value of remote called method `reportLocation` which is `bool`.

In the object method implementation of object `NETWORK_HEALTH`, it is defined that every 20 m (line 33) the current values of individual voltages are collected (line 35) and analyzed (not included) to monitor system health.

The object method implementation of object `MONITOR` includes the implementation of its constructor method `monitor.start` (lines 37 - 38) and its exported method `monitor.reportLocation` (lines 39 - 40).

This application is used as a running example throughout this chapter. It is compiled and evaluated on an actual sensor network as well as on TOSSIM.

2.4 Object Maintenance Algorithms

To support EIP abstractions, the main question is how physical state, events, and activities can be uniquely and identically mapped into objects despite of distribution and possible mobility in the environment. This section gives extensive answers.

While all EIP objects have the same declaration syntax and programming interface, underneath the common API, EIP supports three different implementations of objects, namely, *event objects* (created for mobile events defined as those that dynamically change their geographical locations), *region objects* (mapped to static or slowly moving regions), and *function objects* (not mapped to an environmental element).

To alleviate the programmers burden, the EnviroSuite preprocessor can automatically determine the best category for each object based on the keywords used in the `object_condition` statement. Conditions defined in terms of volatile measurements (such as motion sensing) typically give rise to dynamic contexts with rapidly changing node membership, which are more appropriately implemented as event objects. In contrast, conditions defined in terms of slowly changing measurements (such as temperature) result in more stable groups that can be implemented as region objects. Taking sensor type into account therefore allows the preprocessor to make intelligent guesses about the most appropriate group management protocols to use for object implementation. The programmer is allowed (although not required) to annotate the object as event or region object, overriding the intelligent guess of the preprocessor. An incorrect annotation, however, will result in impaired performance. Function objects are similar to region objects, except that they do not interact with the physical environment. In the following, we describe the three different object maintenance protocols, which determine how and when to form the object context, what group management protocols are involved, where to execute object code, and how to compute object attributes.

2.4.1 Event Object Maintenance

Typically, event objects are created dynamically in response to environmental events that may be mobile and usually fast moving. (A compile-time warning is generated if a static instance is declared for such objects.) In the current implementation and in the discussion below, only localized events are supported. By a localized event, we mean those with a geographically limited sensory signature, such as moving vehicles. We call such localized events, *targets*. Supporting events with a large signature that move quickly is challenging because of the high overhead. However, we do support slowly moving large-signature events as described in region objects.

The core component of our event object implementation is the *multi-target group management protocol* (MGMP). When the condition statement of an event object evaluates to true in a new contiguous region, MGMP creates a new globally unique address, *object ID*, and associates it with the geographically contiguous group of sensor nodes which sense the environmental event. The movement of the contiguous region associated with the event results in dynamic changes to group membership. The protocol ensures that the same object ID is maintained for the event object despite mobility and membership changes, so that it can always be addressed via its uniquely assigned object ID. Dynamically created event objects are aware of their ID and must explicitly send it to other objects if they want to be contacted. Observe that the internal details of MGMP are transparent to the programmer. From the perspective of application code, the only visible effect of MGMP is the dynamic creation and deletion of object instances (in response to environmental conditions). These instances encapsulate aggregate object attributes as defined by the programmer.

Internally, MGMP elects a leader in each object context to maintain a persistent and unique object ID, collects raw data from group members in the context, performs aggregation functions on the leader to compute object attributes, and coordinates computation and actuation tasks as defined in object methods.

In the following, we discuss how MGMP maintains object uniqueness (one-to-one mapping of external events to logical objects) and object identity (immutability of the mapping function) for fast moving targets.

State Machine Representation

MGMP treats each node as a state machine. The sensor network around an environmental event might have the state distribution shown in Figure 2.5. It should be noted that although we use circles to indicate sensing areas, we do not assume sensing areas are circular.

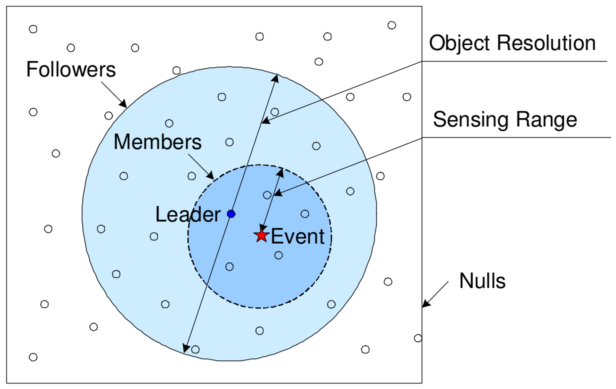


Figure 2.5: States of nodes around an environmental event

All nodes sensing an event constitute the *member* set. A single *leader* is elected by MGMP among the member set. The leader sends periodic *heartbeats* to nodes within half an `object_resolution` (default half is two times the sensing range) away from itself to claim its leadership and to inform them of the existence of the event. Note that the sensing range can be statically derived from the sensor characteristics, and, if the event is detected by a combination of multiple sensors, the shortest one is used. Heartbeats are disseminated through limited flooding, and later on, members communicate to the leader through reverse paths of flooding. The period of these messages, called the *heartbeat period*, is one of the key parameters of MGMP. As we show in the evaluation section, this period can be chosen automatically from a high-level specification of the maximum abject creation latency.

All nodes that cannot sense the event themselves but know of its existence through received nearby leader heartbeats are said to be in the *follower* state. All MGMP control messages are transmitted to nodes within half `object_resolution` away from senders. Thus, half the `object_resolution` must be no less than two times the sensing range, since nodes within the same sensing area must communicate with each other to agree on a single leader. The minimal tolerable object resolution is therefore four times the sensing range.

At any point of time, a node stays at a single state from a set of states S_s:

$$\begin{aligned} S_s \;=\; & \{Null, Follower, Member, NewCandidate, LeaderCandidate, \\ & Leader, ResigningLeader\} \end{aligned}$$

To make MGMP suitable for sensor devices with limited computation and storage ability, we allow each node except leaders to maintain only one object/object ID to reduce algorithm complexity both in time and space. For instance, a node cannot act as `member` of two different objects/object IDs. Figure 2.6 depicts the general state machine algorithm of MGMP without providing details of associated objects/object IDs.

Maintaining Object Uniqueness

Object uniqueness can be compromised in several cases. The first is at the time when a new event causes the creation of a new object. Multiple object IDs for one event may be created since there is no agreement on a single leader initially. To solve the problem we employ a *delayed object creation* mechanism, which delays the creation of a new object by an amount called the *candidate period*, until we are of high confidence that the group of nodes has elected a single `leader` node. In this mechanism, `null` nodes, when sensing an event, transit their states to `newCandidate` and begin to send periodic `CANDIDATE` messages at the

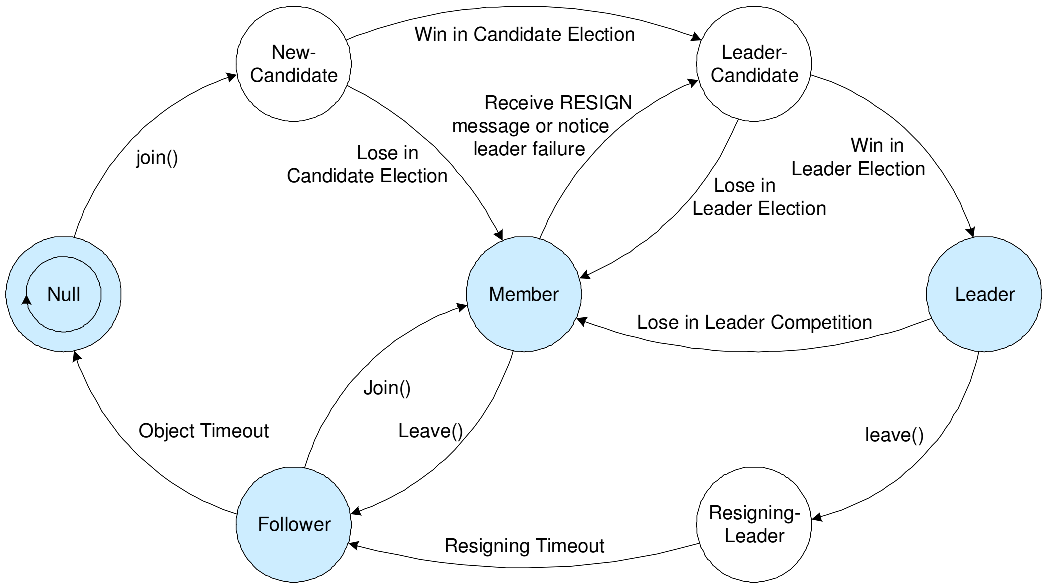


Figure 2.6: State machine in MGMP

heartbeat period, containing sequence numbers and their own node ID. To prolong system lifetime, instead of using the fixed heartbeat period, we can enhance energy balancing by using a dynamic period inversely proportional to remainder energy of nodes. Hence, nodes with a higher energy will become candidates first and will have a higher chance of being elected. In the case of re-transmissions, candidates with a higher energy can back-off less, hence having a higher chance of successfully claiming leadership. The node with the smaller sequence number or, if sequence numbers are equal, with the bigger node ID is forced to quit from the `newCandidate` state and transition to state `member`. This procedure is called *candidate election*, which finally results in only one node at the `newCandidate` state. After a given delay (namely, the candidate period) this node transits to the `leaderCandidate` state. The candidate period is measured in the number of periodic `CANDIDATE` messages sent before one `newCandidate` node can transit to the `leaderCandidate` state. The candidate election algorithm ensures a single `leaderCandidate` in the absence of message loss. Even if messages can be lost, by increasing the candidate delay, a single `leaderCandidate` can be generally guaranteed since the possibility of consecutive message loss is small. In the evaluation section, we determine a good choice for the candidate delay, such that the programmer need not be involved in the decision.

The next problem that compromises uniqueness occurs during leader re-election. When tracked events move out of the current leaders sensing ranges, these leaders must handover their leadership to other nodes, which is called *object migration*. Object migration, especially frequent object migration caused by fast moving events, challenges the maintenance of object uniqueness. MGMP solves this problem by introducing the `follower` state. Through heartbeats from leaders, `follower` nodes know in advance the event objects associated with incoming events. When these nodes come to sense these events, they join the existing objects as `member` instead of creating spurious objects. It was shown in [2] that this mechanism is successful in maintaining object uniqueness as long as object velocity is below some maximum limit.

The third case that challenges object uniqueness is when multiple events of same signatures become closer than defined object resolution, or even cross each others path. To simplify the situation, we assume that event crossing does not coincide with event disappearance. In the previous cases without event crossing, the delayed object creation mechanism and the introduction of the `follower` set ensures object uniqueness. Here, we need only to prevent accidental object termination during event crossing, so that object uniqueness is maintained. The leadership handoff mechanism used in MGMP prevents object termination as long as the object is maintained by one `leader` node and at least one `member` node. Thus, the key in maintaining object uniqueness during event crossing is to balance `member` nodes between merging objects to assign at least one `member` for each object, which is detailed below.

To show the *member balancing* mechanism, Figure 2.7 depicts part of the state machine, which describes how `member` nodes choose their corresponding objects. $\texttt{Member}_x$ denotes `member` state with object ID x. A simple way to balance Member nodes is to divide `member` nodes based on leader position. When a `member` node receives heartbeats from multiple objects, it chooses to join the one with the nearest leader since there is a higher possibility that this node is sensing the same event as that leader. However, such division is not accurate since leader positions are not identical to event locations. It is also possible that a `member` node is actually sensing the same event as the farther leader. For this reason, a new state called `freeMember` is introduced into the state machine. The continuous reception of n continuous heartbeats from `object 2` can transit $\texttt{member}_1$ to `freeMember` and then to $\texttt{member}_2$ even if the last heartbeat was from a closer leader (`object 1`). The introduction of `freeMember` allows wrong choices to be corrected, thus ensuring correctness of member balancing. The member balancing mechanism prevents object termination successfully, therefore enhancing object uniqueness in the third case.

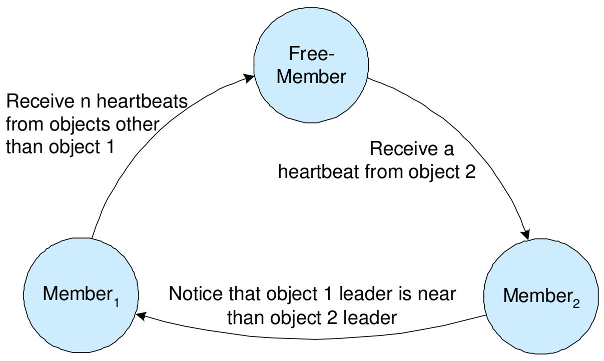

Free-Member
Receive n heartbeats from objects other than object 1
Receive a heartbeat from object 2
Member$_1$
Notice that object 1 leader is near than object 2 leader
Member$_2$

Figure 2.7: Member balancing mechanism

Maintaining Object Identity

While object uniqueness refers to maintaining a single object representation for each external target, maintaining object identity refers to keeping the *correct* association between external targets and their representing objects. In the case where events with same signatures are closer than one object resolution, an extra mechanism is required to maintain identity since member balancing only ensures object uniqueness. We make the default extra assumption that targets tend not to change direction abruptly. This assumption, for example, allows disambiguation of crossing targets based on their path. The default assumption can be customized by programmers if needed.

EIP keeps a record of the recent trajectory of each target (storing it within its representing object). To reduce system overhead, instead of using position information of group members to estimate target locations, we take leaders positions as an approximation. When transferring leadership, each leader also transfers its maintained history of the last $n{-}1$ old leaders positions plus its own. When two events E_1 and E_2 cross each others path, each object leader is able to receive the heartbeat from the other. Each object leader marks itself by the concatenation of the old object ID (O_1) and the new object ID (O_2) as its temporary object ID (O_1O_2). The two leaders exchange their event trajectories such that each remembers both. After separation, a disambiguation algorithm is used, based on recorded history and current locations to chose the ID assignment most consistent the default (straight path) assumption.

2.4.2 Region and Function Object Maintenance

Region object maintenance differs from event object maintenance since region objects are associated with a relatively fixed set of nodes. What we implement for region object maintenance is a spanning-tree based information collection structure described in [37]. Like event objects, the details of region object maintenance are transparent to the programmer. The application code is only aware of the object and its encapsulated aggregate attributes.

When a region object is initialized (statically at system deployment time or dynamically, depending on whether a static instance is declared), a default leader node disseminates tree construction requests to the object context with a running hop-count initialized to zero. Requests are flooded outward with hop-count incremented at every intermediate hop. After receiving tree construction requests, nodes establish multiple reverse paths towards the sending nodes. As a result, a multi-parent diffusion tree is constructed with the leader residing at the root. Spanning tree construction stops when nodes are reached that do not satisfy the region object condition statement. Such nodes become the outer boundary of the tree and serve a role similar to followers in event objects. If these nodes ever satisfy the condition statement they become members and recruit other followers for which the statement is not satisfied. Also, if tree leaves cease to satisfy the object condition, they truncate themselves from the tree and become outer boundary nodes. Hence, membership of the tree can change slowly over time. Measurements needed to compute object attributes can flow up the tree from members towards the leader and get aggregated along intermediate hops. We do not provide details of aggregation algorithms here, since similar mechanisms have been described in previous literature such as directed diffusion [44] and TAG [64]. Our contribution lies in the uniform programming abstractions presented on top of such mechanisms.

One aspect where our region object maintenance algorithm differs from previous work is that we automatically migrate the root of the aggregation tree to the location that minimizes communication and aggregation overhead, as well as to a higher energy node, periodically escaping energy depleted regions. This load balancing flexibility is made possible in our programming model since we implement the program inside the network, alleviating external bottlenecks. After each migration, a (possibly partial) tree reconstruction is done to form a new spanning-tree rooted in the new host node.

The introduction of region objects enables EIP to support not only tracking functions, but also region monitoring functions such as contour finding and system health monitoring, thus making EIP applicable to a broader set of applications.

Function objects are quite similar to region objects except that there are no object contexts and object attributes in function objects. There is no need for object context maintenance and object attribute collection since function objects do not interact directly with the physical environment. The leader of a function object always migrates to the gravity center of all other objects which have recently communicated with the function object through IOC or global variable access.

Like in event objects, leaders in region objects and function objects are responsible for object method execution.

2.5 Implementation

In this section, we take nesC, the most popular language in sensor network area, as the general language that implements EIP. EIP object declarations (defined by programmers) and object methods (assumed to be written in nesC by programmers) are to be automatically translated by the EIP preprocessor into a whole nesC application by selecting and integrating primitive algorithms provided in EIPLib. This section describes how we design EIPLib to simplify the work of the preprocessor and how we implement the preprocessor with the help of EIPLib. Although the implementation details are specific to nesC, most design decisions we make in this section are portable to other languages.

All nesC applications consist of a set of *components*. A component provides and uses *interfaces*, as defined in the components *provides* and *uses* clauses. An interface describes the parameters of a set of *commands* and *events*. There are two types of components: *modules* and *configurations*. Modules provide application code, implementing one or more interfaces. Configurations connect interfaces used by components to interfaces provided by other components. The action of connecting component interfaces is called component *wiring*. It is the main mechanism for building large applications from smaller modules. Wiring is done at compile time, and offers no run-time overhead. We use wiring extensively to connect application components to components implemented by our language libraries.

2.5.1 EIP Support Library (EIPLib)

EIPLib contains a series of primitive algorithms to be used by the preprocessor to build comprehensive applications in nesC, currently including: sensor data processing algorithms (condition library), aggregation algorithms (aggregate method library), object maintenance algorithms and inter-object communication protocols. It also contains higher level services as potential consumers of primitive algorithms, including: object context determination components, object attribute collection components and object method execution components.

Each condition such as `temperature()` and `vehicle_sound()` is associated with a sensor data processing algorithm in EIPLib, which returns processed sensor outputs either as a meaningful value or a boolean either immediately or in a phase-splitting way. However, the association and ways of accessing are hidden in the preprocessor and programmers are only aware of available condition names and their purposes. Also, each aggregation method such as `AVERAGE` is associated with an aggregation algorithm which implements the method. In the current version, object maintenance algorithms contain separate implementations for three object categories: event objects, region objects and function objects. As stated in the beginning of Section 2.4, the object categories are transparent to programmers and are determined by the preprocessor based on the `object_condition` statement. (Advanced APIs are provided for sophisticated programmers to override default rules.) Inter-object communication protocols provide supports for maintaining links between dynamic objects, which is required to implement IOCs and global variable access. All these primitive algorithms are implemented as nesC components with standard interfaces.

Object context determination components determine whether the current node should join some object context based on object declarations. Object attribute collection components collect raw object attributes from member nodes, apply aggregation methods to form aggregate attributes in leader nodes, and support access to aggregate attributes. Object method execution components are responsible for executing object methods in leader nodes whenever corresponding objects exist. These higher level components are also implemented in the form of nesC components, yet differ from usual nesC components in many ways, including:

- They are not pre-wired since object declarations and object method implementations are not available until compile time. Wiring is left for the preprocessor so that primitive algorithm components may be freely selected and wired into higher level components to construct any EIP applications defined by programmers.

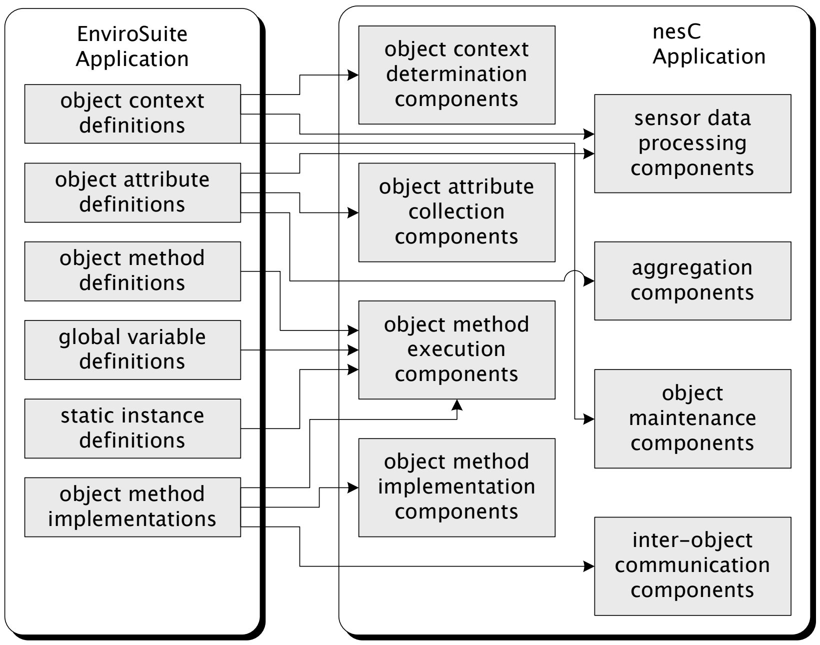
- They contain special clauses that are recognizable only by the preprocessor. In many cases, such clauses are necessary to guide language translation. For example, the configuration of object context determination components may include a special clause ({`__ES_COMPONENTS`}) which indicates the position where necessary sensor data processing components are to be listed by the preprocessor. These clauses greatly simplify the implementation of the preprocessor by giving some hints.

The hierarchical structure between primitive algorithms and high level components is also critical. In this structure, various configurations can be achieved by changing only the high level components while other components can remain unchanged, thus reducing the complexity of the preprocessor.

2.5.2 EIP Preprocessor

The EIP preprocessor is essentially a translator that takes EIP code as input and outputs desired environmental monitoring applications in nesC, which then can be compiled by a standard nesC compiler and uploaded to the motes. It is implemented in Perl, a language with powerful built-in support for text processing. Current implementation contains 1533 lines.

EIP application code consists of two parts, object declarations and object method implementations. The detailed translation of both parts is illustrated in Figure 2.8.



EnviroSuite Application
object context definitions
object attribute definitions
object method definitions
global variable definitions
static instance definitions
object method implementations
nesC Application
object context determination components
sensor data processing components
object attribute collection components
aggregation components
object method execution components
object maintenance components
object method implementation components
inter-object communication components

Figure 2.8: Translate an EIP application into a nesC application

The preprocessor analyzes object declarations line by line, making corresponding configurations and integrations. As depicted in Figure 2.8, for object context definitions, it identifies all conditions, locates the corresponding sensor data processing components by searching condition library, which lists all condition names and the corresponding implementations, wires them into object context determination components. A feature of the preprocessor is that it automatically determines the best category for each object based on these conditions and integrates the corresponding object maintenance algorithms.

For object attribute definitions, besides identifying conditions and wiring corresponding components into object attribute collection modules, the preprocessor also wires aggregation components. Additional work includes setting attribute refreshing timers based on `attribute_freshness` definition and validating resulted aggregate attribute based on `attribute_degree` definition. Based on object method definitions, the preprocessor wires the implementations into object method execution components. The preprocessor also copies global variable definitions into object method execution components and enables remote access to global variables by implementing local read and write commands, which respond to received remote calls. For each static object instance, the preprocessor randomly selects a node as the default leader, which initially executes the main object function, and migrate the leader to a more power-efficient position later.

The preprocessor also filters object method implementations for keywords, translating `ES_GETATTRIBUTE` into command calls to object attribute collection components and `ES_IOC`, `ES_IOCRESULT`, `ES_READ` and `ES_WRITE` into command calls and event handlers of inter-object communication components.

As seen above, the EIP preprocessor successfully bridges between low-level implementations in EIPLib and high-level abstractions exported by EIP by making several intelligent steps that are transparent to the programmers: selecting sensor data processing algorithms; automatically identifying object categories and applying corresponding maintenance algorithms; and automatically collecting and aggregating attributes from multiple nodes.

Observe that, one clause in an EIP application may result in multiple changes in higher level components, and one higher level component from EIPLib may be changed multiple times by multiple EIP clauses, which means the preprocessor may need to change the same file in EIPLib repeatedly. Considering such phenomenon, instead of creating corresponding new code line by line, we store the resulting changes in a hash of hashes, so that already changed code can be further changed easily. The hash of hashes stores, for each file and each special clause such as {`__ET_COMPONENTS`}, their corresponding nesC code. Only after analyzing the entire EIP application, the preprocessor changes files from EIPLib based on the resulted hash of hashes. The storage space needed by the hash of hashes may be very large. However, we consider it acceptable since the preprocessor runs on a PC and therefore does not have severe storage constraints.

2.6 Performance Evaluation

This section provides a detailed quantitative analysis of EIP. We begin by evaluating the performance of a series of micro-benchmarks on simulators, which analyze the primary features of EIP: object uniqueness and identity maintenance, and inter-object communication support. The first set of benchmarks tests object uniqueness and identity management during object creation, object migration and object crossing (which is the most challenging case). The second set of benchmarks tests inter-object communication. We then move to real platforms to evaluate the performance of a surveillance system built using the EIP framework. Both tracking performance and monitoring performance are evaluated to demonstrate event objects and region objects. The evaluated system is the one described in Section 2.3. Its abbreviated code is given in Figure 2.4.

2.6.1 Performance of Object Operations

To evaluate the performance of primitive object operations, we choose TOSSIM since EIP produces real nesC code for motes and TOSSIM can emulate the execution of the real code on the motes without the need for deployment. The radio model simulated in TOSSIM is almost identical to the 40 Kbit RFM-based stack on the motes. To control per-hop message loss at the packet level we added an external program component. We focus on fast-moving objects (event objects), since their real-time maintenance offers the most challenge to the EIP infrastructure.

In our emulated experiments, we set the sensing range to 100 feet (approximately 30 meters). Current sensor devices such as the micropower impulse radar [5] can detect objects up to 50 meters away. Radio range is set to 300 feet. Current sensor network products such as the Mica2 and Mica2Dot motes [15] have a maximum outdoor radio range of 500 feet to 1000 feet under ideal conditions when sending with full power. Sensor nodes are placed on a grid 100 feet apart.

Experiment 1 - Object Creation

EIP associates a logical object with each physical event. It is critical that such association should be done as soon as possible to reduce the inconsistency between the physical world and the logical world exported by EIP. In the first experiment, we measure object creation delay, defined as the difference between the time the first node senses an external stimulus and the time an object ID is created for the corresponding object.

The external entity tracked, in this case, is a vehicle. The tracking code is given in Figure 2.4.

The delay of object creation is decided by both the candidate period, which indicates how many candidate messages must be sent before creating objects, and the heartbeat period, which determines candidate message intervals. In the following we show the experimental data that allow these parameters to be selected automatically by EIP from a high-level specification of the maximum tolerable object creation delay. Figure 2.9 plots object creation delay versus heartbeat period for different candidate periods.

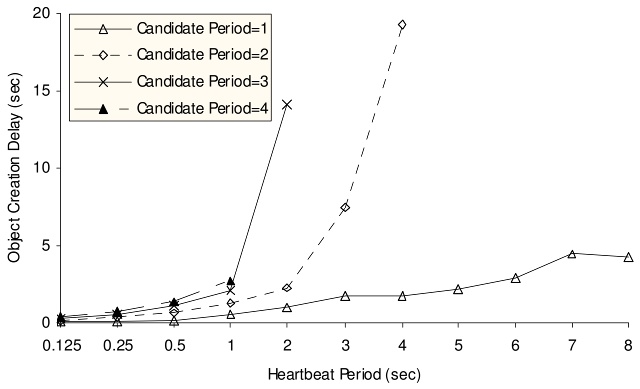


Figure 2.9: Object creation delay for varied heartbeat period and candidate period

From Figure 2.9 we observe that object creation delay increases with the increase in both the candidate period and the heartbeat period. The plots show only those points for which a non-zero number of objects are created. A candidate period of 1 performs best in terms of object creation delay. However, it is undesirable since it causes spurious objects at higher heartbeat periods as stated below.

The candidate period and the heartbeat period affect not only object creation delays but also object uniqueness. Figure 2.10 shows the impact of the candidate period and the heartbeat period on object uniqueness by plotting the average number of created objects. Ideally, only one object should be created per experiment, since the only target is deployed.

Figure 2.10 shows that with shorter heartbeat periods, candidate periods 2, 3 and 4 perform similarly. However, longer candidate periods result in a longer object creation delay, so that when the heartbeat period exceeds a certain threshold, objects cannot be formed in time before the vehicle moves out of their sensing ranges. Figure 2.10 shows that for candidate period 4, it is difficult to create objects after the heartbeat period exceeds 2 seconds, while for candidate period of 2, objects can be created up to a heartbeat period around 5 seconds. We therefore choose 2 as the default candidate period in EIP. A longer candidate period should be chosen in the presence of message loss.

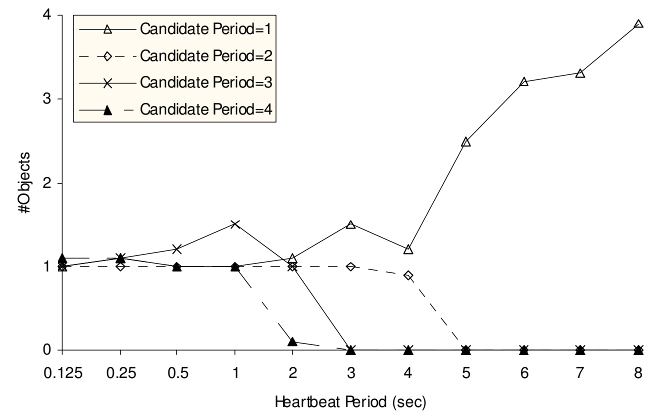


Figure 2.10: Number of objects created for varied heartbeat period and candidate period

Given the default candidate period (of 2), the object creation delay can be chosen anywhere from a small fraction of a second to multiple seconds depending on the choice of heartbeat period, as shown in Figure 2.9. The programmer should therefore specify a maximum tolerable value of object creation delay. This specification stems easily from application domain knowledge. For example, in a vehicle tracking application, a delay of 1-2 seconds between vehicle entry into the field and the creation of a corresponding event object is quite tolerable. EIP then uses Figure 2.9 to compute the corresponding heartbeat period. Observe that a smaller heartbeat period implies more communication, more energy consumption, and consequently a shorter lifetime. Hence, a trade-off exists between system responsiveness (object creation delay) and lifetime.

Experiment 2 - Object Migration

The core part of EIP is to uniquely and identically map physical events to logical objects. In this experiment, we reveal how fast object migration could be performed without breaking object uniqueness and identity. Object migration is caused by the movement of associated events. Hence, from the perspective of applications, the velocity limit of object migration is more meaningfully expressed by the maximum tolerable event velocity. It is defined as the maximum velocity of events, which can be uniquely and identically mapped to logical objects. Observe that for a given maximum object migration speed (in hops per second), the corresponding maximum event velocity depends on the radio range (distance per hop). The data presented below is for the range parameters mentioned in Section 2.6.

We explore several factors, which affect maximum tolerable event velocity, including heartbeat period and object resolution (in multiples of sensing range). As is shown in Figure 2.11, the maximum tolerable velocity increases when the heartbeat period decreases, since a shorter heartbeat period results in a shorter leader re-election delay and thus a higher trackable velocity. This trend is reversed when heartbeat period becomes short enough to cause message loss or congestion as shown in Figure 2.11 (for a half object resolution of 2 sensing ranges) when heartbeat period falls below 0.5 s. Increasing object resolution has positive impact on the maximum tolerable velocity since a bigger set of followers allows the vehicle to go farther without causing new object creation. Similar results were reported in [2]. We stress, however, that results reported in [2] were obtained from algorithm simulation in GloMoSim. In contrast, results presented in this chapter test the performance of actual nesC code generated by our EIP preprocessor for the application in Figure 2.4.

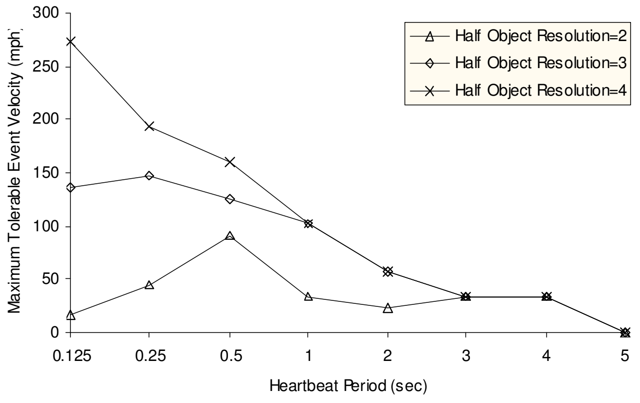


Figure 2.11: Maximum tolerable event velocity for varied heartbeat period and object resolution

Next, we evaluate how robust the object uniqueness guarantee is against message loss during object migration. TOSSIM does not provide message loss models at the packet level. Thus, we add a simple external program to control per-hop packet loss ratio. Figure 2.12 depicts the average number of objects formed per run as a function of target velocity in the presence of different degrees of packet loss. As before, the ideal number should be 1 object per run.

From Figure 2.12, we see that EnvoroSuite can completely tolerate a 10% loss ratio since we get similar results to those with 0% loss ratio. EIP can also tolerate a loss ratio of up to 30% when event velocity does not exceed 68 mph. Larger velocities or loss percentages, however, cause spurious objects to emerge. Observe that at a very high event velocity, the number of formed objects decreases again, which might seem like an anomaly. The explanation lies in that very high speed objects do not have enough time to form in the

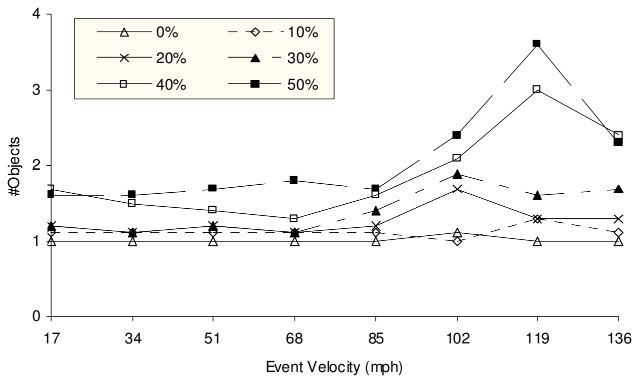


Figure 2.12: Number of objects created for varied event velocity and per-hop packet loss ratio

first place.

Experiment 3 - Object Crossing Performance

Next, we explore the efficacy of EIP in maintaining object uniqueness and identity when two objects of the same sensory signature (i.e., fulfill the same `object_condition` statement) cross paths. In this experiment, two vehicles are moving straight along crossing diagonals with the same speed of 24 mph. The diagonals cross in the center of the field. However, these objects may not start at the same time, and hence may not reach the crossing point together. We vary their relative start times to vary the shortest distance reached between the two objects at the crossing point (which we call, the *crossing distance*). We show the percentage of runs where object uniqueness and identity are maintained as a function of crossing distance. As shown in Figure 2.13, object uniqueness and identity are ensured in most cases even when the two targets cross the center point at the same time (crossing distance is 0).

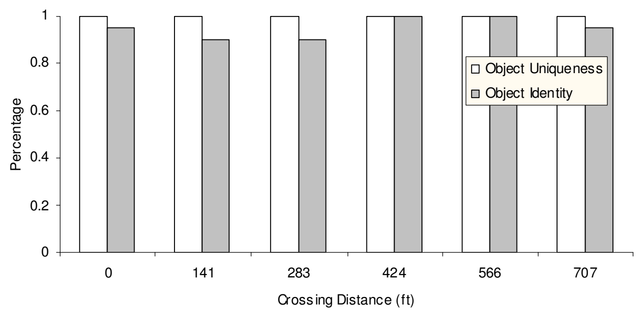


Figure 2.13: Achieved object uniqueness (white) and identity (shaded) for varied crossing distance

Each bar in Figure 2.13 represents the average of more than 10 runs. The tracked trajectory for one run with crossing distance 0 is shown in Figure 2.14. After passing the center point, although object identity is lost for a while, the system successfully recovers from the confused state after accumulating enough history.

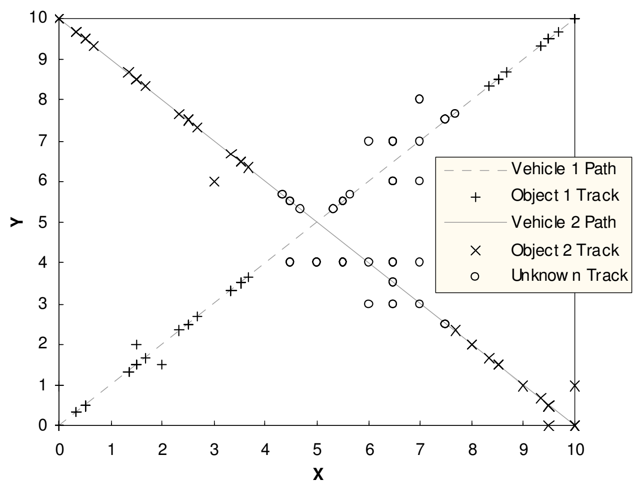


Figure 2.14: Reported target tracks with crossing distance 0

The results prove the relative success of our adopted direction disambiguation algorithm. It also shows that defensive programming is advisable. While we elevate the level of abstraction to that of objects representing environmental elements, the programmer should expect such objects to be occasionally confused. The application code may chose to implement its own disambiguation on top of EIP object IDs.

Experiment 4 - Inter-object Communication

Programming for communication and coordination between objects becomes very simple by using IOC and global variables. In this experiment, we evaluate a vehicle counting application, which counts the total number of vehicles in a global variable, to analyze the performance of inter-object communication. As seen in the code from Figure 2.4, whenever a vehicle appears, the global variable `vehicleNumber` is increased by one through an `ES_WRITE` call from the corresponding `vehicle` object.

In this scenario, four vehicles enter the coverage field one by one, maintaining the same speed of 35 mph and thus the same distance. The first one goes straight from (-1, 1) to (16, 1); the second from (-5, 5) to (16, 5); the third from (-9, 9) to (16, 9); the last from (-13, 13) to (16, 13).

Figure 2.15 plots the counter values as a function of time in this application. Input represents real numbers of vehicles. Output represents the counting results achieved by the application. Delay between the input and output curves represent the end-to-end performance of remote object invocation. These delays reflect the sum of object creation delay and inter-object communication delay.

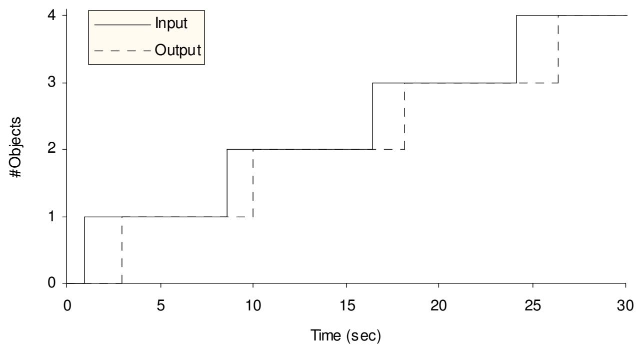


Figure 2.15: Vehicle counting application results

Service Name	Code Size (KB)	RAM Usage (B)
Sensing Data Processing	10.8	18
Event Object Maintenance	25.6	75
Region and Function Object Maintenance	28.5	44
Inter-object Communication	15.0	45
Other Service (Aggregation, etc.)	25.1	90
Object Method Components	6.0	47

Table 2.2: Code size and RAM usage of different services

2.6.2 A Surveillance System

Finally, we test the complete surveillance application written in EIP, described in Section 2.3. This surveillance system tracks all in-field vehicles, counts their number and monitors system health at the same time. The EIP code of this application can be translated by the EIP preprocessor into a nesC application. Emitted nesC code size and RAM usage of different services in the translated application is listed in Table 2.2.

Table 2.3 compares EIP code and emitted nesC code of the same application in terms of module number, code length and size. The code size of the nesC version gives a good estimation of required programming effort if the whole system is to be programmed directly in nesC. As is seen from Table 2.3, the code size of the nesC version is more than ten times of that of the EIP version. Thus, the estimated programming effort with EIP is roughly an order of magnitude less. The result reflects the efficiency of EIP compared with node-based languages, such as nesC.

	Module Number	Code Length (lines)	Code Size (KB)
EIP Version	3	218	5.9
nesC Version	12	3692	111.0

Table 2.3: Code comparison of EIP version and nesC version

Tracking Performance

In this experiment, we evaluate the efficiency of EIP in terms of tracking performance and power consumption by comparing it to a simple baseline. This baseline is to plot the trajectory of a tracked target at a base station located in $(0, 0)$. In the EIP implementation, members, who are sensing the target, report to the current leader their own positions every 0.5 seconds. The leader aggregates these positions and reports the average to the base station twice per second. The baseline has a simple implementation of the same application. Each node that senses the target sends its own position to the base station every 0.5 seconds. The base station averages received positions twice per second. In both the EIP version and the baseline, a minimum aggregation degree of 2 is enforced to reduce false alarms.

The actual testbed for this experiment consists 40 Mica2 motes laid out in a $10{\times}4$ grid with integer (x, y) coordinates ranging from $(0, 0)$ to $(9, 3)$. The goal is to track a rectangular object, 1 square grid in size, moving straight along the middle of the longer axis, with a speed of 0.5 grid per second. This testbed does not take into account errors in localization and time synchronization services. To ensure enough tracking accuracy for real applications, we require that localization errors not exceed half grid and time synchronization errors be kept in the order of ms. Many existent techniques support such precision.

Figure 2.16 compares the target trajectory obtained by the EIP application to the one resulting from the baseline. Some tracking error is seen because our sensor devices have no notion of proximity to the target. As shown in Figure 2.16, the EIP version has a smaller average tracking error compared with the baseline although it introduces a little more variability. The underlying reason is that in the baseline, position reports from nodes may not be in order when they arrive at the base station, since they may have traversed multiple hops, which results in more inaccuracy.

Figure 2.17 depicts the number of packets sent or forwarded by each node in a slice of the network over the duration of the experiment (where X, Y is the coordinate of each node). Each bar in this figure represents the average of 15 runs to ensure a statistical significance at the 0.05 level. The number of packets is important as it is proportional to power consumption. It can be seen that the EIP version achieves its comparable tracking performance with much less power consumption in terms of the number of transmitted packets. Hence, our tracking algorithms are more energy-efficient.

In the baseline test, most packet transmissions occur on nodes with Y coordinates between 1 and 2 since only these nodes can forward the packets to the base station. The nodes with smaller X coordinates

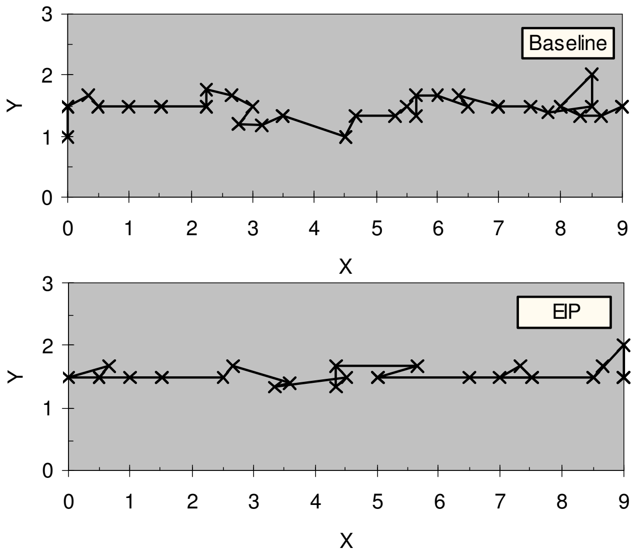


Figure 2.16: Tracked target trajectory comparison

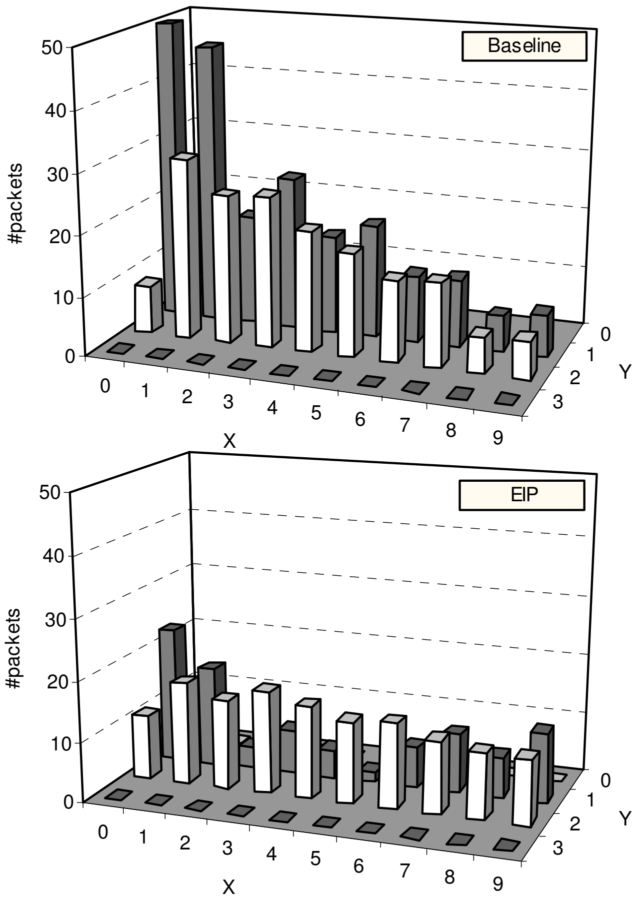


Figure 2.17: Transmitted packet number comparison

in the baseline send much more packets than those in the EIP version since each node sensing the target sends packets directly to the base station located in $(0, 0)$. Hence, a greater number of packets have to be forwarded by nodes with smaller X coordinates. In the EIP version, position reports are aggregated locally by leaders, amounting to much fewer packets forwarded to the base station.

Monitoring Performance

In this experiment, we utilize the `NETWORK_HEALTH` object coded in Figure 2.4 to monitor the health of the network by collecting information on nodes that are alive and their remaining power. Alarms will be sent out if a big portion of the network is dead or lacks power.

We carry out this experiment on a network of 27 XSM motes [74] deployed in a grassy field. The system performs the function of vehicle tracking as well as health monitoring. For system health monitoring, the `NETWORK_HEALTH` object is determined as a region object by the EIP preprocessor, thus a multi-parent spanning tree is automatically constructed at object initialization to collect power information of each node every 20 minutes. The system is tested for several hours. Figure 2.18 depicts the collected power information, where the black bars represent initial voltage reported by the region object, the grey ones show voltage reported after 20 minutes and the white ones shows voltage reported after 40 minutes. Node 0 is the base node, which consumes the most power.

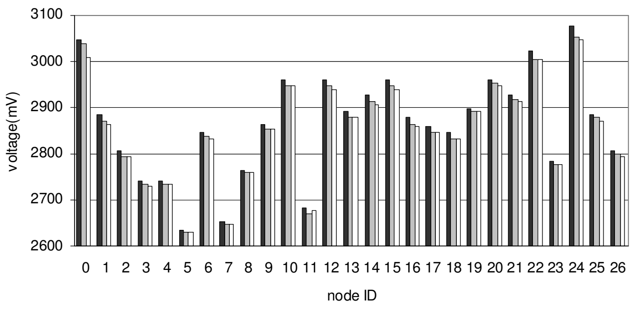


Figure 2.18: Power level of each node for different times

2.7 Discussion

To better define EIP, this section supplies a discussion on features and limitations of the EIP primitives.

First, EIP is not a replacement to other emerging programming paradigms such as group-based primitives, database-centric abstractions, event-based systems, and virtual machines. We do not argue for a single approach to the exclusion of others. The most appropriate abstractions are often a personal choice that depends on subjective programmer preferences as well as application specifics. Ultimately, it is the availability of multiple programming alternatives that induces more software development. EIP is therefore presented and evaluated for its own merits, and not as a substitution for other high-level paradigms.

Second, EIP is not a programming language in itself. Instead, EIP extends other programming languages with environmentally-immersive programming primitives. This extension takes two different forms. First, the programmer is allowed to define and use variables that summarize elements of a potentially distributed environmental state (such as the average temperature of a region or the current location of a moving target). Second, the programmer may define code that is geographically distributed and associate the time and place of its execution with the occurrence of certain environmental events. Both the aggregate variables and distributed code are encapsulated within simple objects.

Third, EIP achieves expressiveness by seamlessly combining itself with an imperative language. EIP primitives are declarative in nature. They allow programmers to describe what physical entities and which aspects of the entities they are interested in, rather than to specify how the physical entities are detected and tracked and how the aggregate attributes are calculated. EIP does not contain primitives for programmers to specify what to do with the physical entities and their attributes. Therefore, EIP in itself is not capable of specifying a wide variety of applications. But, when combined with an imperative language, say nesC, EIP achieves much better expressiveness. For applications with complex requirements, programmers can specify (in the imperative language) tasks and attach them to the physical entities using EIP scripts.

Fourth, EIP syntax simplifies integration of the underlying middleware P syntax simplifies integration of the underlying middleware services. The whole purpose of EIP is to abstract the distributed aspects of environmental interactions and computation. Behind EIP are essentially a set of middleware services containing the distributed algorithms. Alternative to using EIP syntax, advanced programmers can embed code into their applications and directly invoke the distributed algorithms. However, EIP scripts and the preprocessor make integration of the middleware services much simpler.

Finally, some remarks are in order on the extensibility of EIP. Current EIP supports only certain types of physical entities. This is not a fundamental limitation of EIP itself. Rather, it is due to the limited number of object maintenance algorithms currently available in the underlying library. EIP can be extended to support an even broader set of applications after more object maintenance algorithms are developed. services. The whole purpose of EIP is to abstract the distributed aspects of environmental interactions and computation. Behind EIP are essentially a set of middleware services containing the distributed algorithms. Alternative to using EIP syntax, advanced programmers can embed code into their applications and directly invoke the distributed algorithms. However, EIP scripts and the preprocessor make integration of the middleware services much simpler.

Finally, some remarks are in order on the extensibility of EIP. Current EIP supports only certain types of physical entities. This is not a fundamental limitation of EIP itself. Rather, it is due to the limited number of object maintenance algorithms currently available in the underlying library. EIP can be extended to support an even broader set of applications after more object maintenance algorithms are developed.

2.8 Conclusions

In this chapter, we describe an environmentally-immersive programming paradigm for application developers in sensor networks. We present the design, implementation and evaluation of the basic framework implementing this paradigm. This basic framework successfully exports high-level abstractions, such as objects and inter-object calls. It also implements the required low-level distributed protocols such as sensing data processing, group management and inter-object communication in an underlying library EIPLib, transparent to programmers, thus resulting in a considerable potential to reduce development costs of deeply embedded systems. This chapter describes the first comprehensive design and implementation of EIP abstractions including objects, their attributes, methods and inter-object calls. This chapter also presents the first comprehensive evaluation of the performance of real nesC code generated by the EIP preprocessor from EIP source files.

EIPLib contains only the middleware services that are essential in supporting EIP. In the next chapter, we continue to present another middleware service that provides further support for EIP, a storage system that allows EIP programmers to easily store and retrieve data when real-time connections to base stations are not available.

Chapter 3

EnviroStore: The Storage System

3.1 Introduction

The basic framework presented in Chapter 2 provides support for applications under *connected operation*, which assumes that a sensor network is always connected to a base station that collects the data. Therefore, we provide mechanisms like Inter-Object Call (IOC) to allow timely data upload towards base stations. This assumption is suitable when near-real-time information availability is desirable. Tracking and event notification applications, for example, fall under this category. However, a significant number of applications do not require real-time information. Examples include [59][88][25], just to name a few. They operate under scenarios where deployment of base stations is either impossible or inconvenient. Therefore, there is no direct connection from the deployed sensor network to base stations or the connection is rather intermittent. This is what we call a *disconnected* network model.

In the disconnected operation model, real-time data upload (through IOCs between event/region objects and function objects that sit at base stations) are either unnecessary or impossible. Instead, there is a demand for new abstractions that utilize in-network storage (already available on sensor nodes as flash memory) for EIP objects to deposit in the network its major outcomes, including the aggregate state of external events encapsulated in objects, and the computation results of object methods. Later, the stored data can be collected for post-experiment analysis.

The disconnected model does not preclude sporadic contact with a base station during network lifetime. For example, a user may choose to visit the field periodically for maintenance purposes (e.g., to remove dirt and debris that may occlude light sensor inputs over time). Such visits may be used for opportunistic data upload. The user could carry a data mule device [77] that collects data wirelessly from encountered nodes and dumps these data later to the base station (e.g., a computer in the user's office).

A primary concern of the sensor network in this model becomes that of maximizing effective storage capacity (i.e., minimizing data loss due to flash memory overflow while the network is not connected). Observe that some nodes will record more data than others. This may be due to asymmetry in environmental inputs (e.g., acoustic nodes near sound sources will fill up before those in quiet areas), or due to data-dependent variations in compression ratio of algorithms such as run-length encoding. Data loss can be minimized by migrating data from nodes that are full to those that are not, as well as by exploiting upload opportunities when available.

In this chapter, we present *EnviroStore*, a cooperative storage system for sensor network applications optimized for disconnected operation. EnviroStore employs data redistribution schemes to optimize sharing of network storage. The amount of data that can be stored in the network is also affected by the power consumption of nodes. Naturally, nodes will be unable to record data after energy is depleted. EnviroStore takes into account the rate of energy consumption to avoid depletion-related data loss. Evaluation shows that in networks with a large input data imbalance, EnviroStore can delay the onset of data loss by nearly an order of magnitude. Note that EnviroStore is an independent service, geared to isolate low-level storage concerns not only for applications built upon EIP but also for general sensor network applications. EnviroStore can also be easily configured to be a simple storage system that works under connected operation model.

EnviroStore reflects a change in paradigm for sensor network operation from communication-centric to storage-centric. Indeed, the growing size of low-power flash memory suggests that future nodes will have a much larger storage compared to their communication bandwidth. With the increase in storage capacity, new higher-bandwidth sensing modalities (such as multimedia) will undoubtedly be deployed that take advantage of the extra storage space. This will further exacerbate the communication bottleneck as low-power radio bandwidth does not grow at the same rate as low-power storage capacity. Multi-hop communication will need to be minimized. Physical data collection via data mules will become more common, motivating data services such as those described in this chapter.

A storage-centric paradigm for sensor networks differs significantly from that of traditional distributed file systems. First, it has to be simple and lightweight considering the limited CPU bandwidth and memory of sensor nodes. For example, MicaZ [15], one of the most current platforms, has only an 8MHz 8-bit processor and a 4KB RAM. Second, directories and file names need not be maintained in the sensor network itself. These abstractions are needed only on the collection station. Sensor nodes simply write data to the collection station (via a delay tolerant network). They never read the data they write. The system essentially abstracts away the delay-tolerant network between the data sources and the sink. Third, data redistribution is an essential component of the system to avoid inefficiencies of partitioned storage. A good data redistribution scheme is needed to migrate data from highly utilized to less utilized storage spaces in order to improve overall storage utilization. Finally, sensor nodes may not necessarily be connected, forming multiple network islands. Opportunistic data redistribution is needed not only within, but across such islands. These concerns are addressed in EnviroStore design.

EnviroStore focuses on data balancing techniques to achieve lossless long-term information storage. However, it does not preclude the use of other well explored techniques [24] to further prolong the storage lifetime. For instance, lossy/lossless data compression and archiving algorithms can be easily integrated into the system to better serve the goal of data longevity.

We implemented EnviroStore in nesC under TinyOS and evaluated system performance experimentally on TOSSIM. Our evaluation results demonstrate the effectiveness of this service in improving storage utilization in various application scenarios. Up to an order of magnitude improvement was observed in postponing the onset of data loss.

The rest of the chapter is organized as follows. Section 3.2 presents system design. Section 3.3 presents the details of EnviroStore implementation. Section 3.4 illustrates evaluation results. Section 3.5 concludes the chapter.

3.2 Design

3.2.1 System Model

We consider a sensor network that is normally disconnected from the outside world. The function of this network is to collect sensory data. These data must eventually be moved to a data sink. In our architecture, the sink is a process that runs on a user's PC, identified by a regular IP address and a (well-known) TCP port. This process implements a file service that receives sensor network data and organizes them in a local file system on the PC in accordance with some configuration set-up. The sink typically has Internet access. Hence, data can be uploaded to it via the Internet from a remote network. Alternatively, the sink may have an 802.15.4 interface (or an 802.15.4 mote connected to the serial port). A process reading that interface (or serial port) relays data to the well-known port of the file service for storage.

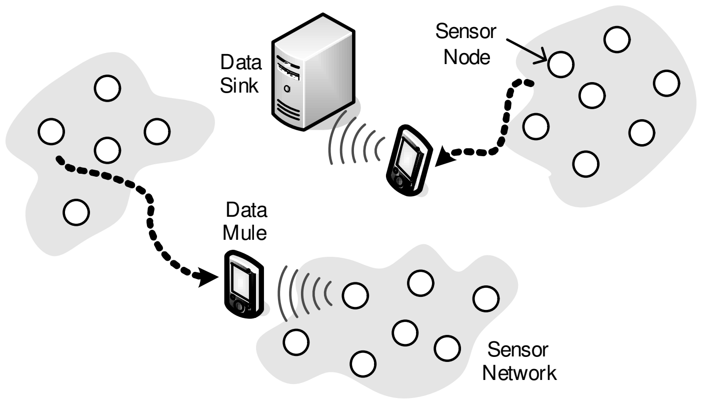


Figure 3.1: System model

Since the sink is normally disconnected from the sensor network, data must be buffered in the sensor network until an upload opportunity arises. Hence, a sensor network storage system is needed. We call this storage system *disruption-tolerant* since it should accommodate sensor network partitions. The disruption-tolerant storage system should, for example, be able to take advantage of data mules to share data across partitioned network islands or offload data to the sink, as is shown in Figure 3.1. Data mules are any (trusted) mobile devices that may come in contact with sensor network islands. For example, they might be handhelds with an 802.15.4 interface and an 802.11 interface. Depending on the application scenario, mules may be eventually able to contact the sink via the Internet or via the 802.15.4 interface.

In the context of sensor network applications that motivate this chapter, only two types of data mules are relevant. The first type represents data mules that intentionally relay data between the sink and the sensor nodes. For example, network maintenance operators who visit the network periodically may also perform data mule functions. Barring unexpected failures, this type of mule is guaranteed to return the data to the sink. For example, it may have Internet access such that it will send collected data to the sink via the Internet when it encounters an access point upon return from the field.

The second type of mule is one whose mobility patterns are independent of data upload needs. For example, consider a library-monitoring study that measures noise levels in different rooms and correlates them with library use[1]. Due to the size of the rooms, not all noise sensors are connected. A librarian performing their normal job functions (that are independent of data collection) can carry a data mule device. Nodes that come in contact with the mule will then opportunistically use it for data upload or redistribu-

[1]This is an actual study planned at the UIUC library to test the hypothesis that optimal use does not require a noise-free environment

Application	Sensor Nodes		Data	Data Sink	
	Static	Mobile	Mules	Static	Mobile
GDI [67][83]	√			√	
ZebraNet [59][48]		√			√
NIMS [6]	√	√			
Macroscope [85]	√			√	
Underwater Sensornet [88]	√		√		
Smart Attire [25]		√		√	

Table 3.1: Application examples

tion. Another example of independent mobility patterns is inspired by a recent experience of the authors, where sensor nodes deployed in a forested area were repeatedly visited by raccoons. The recurrence of these visits suggested the possibility of using members of the local wildlife as data mules. A similar observation was made when an experimental farm was considered for two concurrent telemetry experiments: one was to collect chemical measurements from soil; the other was to track cattle in the same farm using GPS collars. The natural opportunity to design the collar to perform data mule functions for soil sensors suggested a deeper point; as sensors proliferate, the role of opportunistic exploitation of natural mobility in the environment may become more important in network protocol design. Independent mobile data mules are opportunistically exploited by EnviroStore.

The disconnected operation models described above are observed in a large set of other sensor network applications including environmental monitoring, animal tracking, and assisted living. Table 3.1 lists some concrete examples from the recent literature and their suitability for this system model.

3.2.2 System Design

This section presents the decisions we made during the design of EnviroStore to achieve the goals. We will enumerate through the set of mechanisms that cooperatively utilize the storage capacity available at different levels of the system, including in-network data balancing, cross-network data balancing, and automatic data upload.

In-network Data Redistribution

This section focuses on storage management at the sensor node level to maximize the effective storage capacity of the networked nodes. As mentioned earlier, the distribution of the data inputs is not necessarily even among sensor nodes, which calls for data redistribution to improve total storage utilization. An ideal data redistribution scheme should accommodate all data as long as the sum of all node sensory inputs, accumulated over time, is less than the sum of all node storage capacities. A simple solution is to balance storage utilization by offloading data from nodes that are highly loaded to nodes that are not. In a perfectly balanced system, no storage overflow occurs until the total network capacity is exceeded when all nodes reach their flash limit simultaneously. This scenario represents optimal flash usage. Unfortunately, the solution is not energy-efficient. A small change in storage utilization of one node (due to new input) may result in data dissemination to every node in the network even if the source has plenty of storage to accommodate the input. This excessive and unnecessary communication may result in early energy depletion and consequent untimely loss of data.

From an energy saving perspective, it is therefore advantageous not to start offloading data too early. In other words, a *lazy-offload* scheme is preferred. By postponing data balancing until the latest possible time (when flash overflow is imminent), significant energy savings can be achieved. For efficiency reasons, we are interested in algorithms that use local information only. In such algorithms, plateaux must be avoided in the neighborhood to ensure that pathways exist for data flow. In accordance with the above two requirements, a node i, in EnviroStore, decides to offload data only when its remaining storage size (R_i) satisfies one of the following two conditions:

$$R_i = R_{min} \; and \; R_i < R_{TH} \tag{3.1}$$

$$R_i > R_{min} \; and \; R_i - R_{min} < R_{gradient} \tag{3.2}$$

where R_{min} is the minimum remaining storage size within node i's neighborhood (including i), R_{TH} is a configurable threshold to delay data transfer until the remaining free storage is small enough, and $R_{gradient}$ is a configurable parameter introduced to allow for a certain level of local imbalance whose gradient points in the direction of less utilized areas in the network. The first condition indicates that the most-loaded node does not start to transfer data until its remaining storage falls below a threshold R_{TH}. This is to prevent unnecessary energy consumption. R_{TH} should be set big enough to accommodate temporary bursts of input data. The second condition ensures that plateaux do not occur among neighbors, so that data can always flow away from congested nodes.

In our system, nodes exchange periodic advertisement messages sharing free storage information, R_i, within their neighborhood. To conserve energy, such messages are sent at a low frequency (e.g., once per minute). However, to ensure accurate knowledge of neighborhood data to within specified error bounds, extra advertisement messages are inserted when the remaining storage size incurs big changes (of more than R_Δ, the *node advertisement threshold*) since the last advertisement.

If one of the above offload conditions is satisfied, node i should next decide who to send data to. Obviously, i should select the destination from those underloaded neighbors whose remaining storage size is above the average remaining storage ($\bar{R}_i$) of the neighborhood (including node i itself). Always selecting the neighbor with the largest remaining storage turns out to be a bad choice, because a node with the largest R within its neighborhood may be chosen simultaneously by multiple other nodes as their distribution destination. After redistribution, the node may become overloaded and must transfer some of the recently received data back to its neighbors, causing unnecessary consumption of both bandwidth and energy. We call this phenomenon *data ping-pong*. To avoid it, we apply a random function which assigns each underloaded neighbor a non-zero probability (proportional to its remaining storage) of being selected as the redistribution destination.

To further prevent data ping-pong, we should bound the amount of data transfer. Assuming that node i selects node j as the redistribution destination, the amount of data to be transferred from i to j, denoted by D_{ij}, has to satisfy the following condition (not to "overdo" the transfer and reverse the direction of imbalance):

$$R_j - R_\Delta - D_{ij} \geq \bar{R}_j \tag{3.3}$$

Note that, R_Δ is added to account for the inaccuracy in the estimation of R_j on node i, which is caused by the low-frequency of node advertisements. The inaccuracy is bounded by R_Δ (assuming no message loss) since, as previously stated, extra advertisement messages are inserted for changes exceeding R_Δ. Following the same reasoning, after the data transfer, the resulting free storage size of node i should not exceed the neighborhood average, since it may cause data ping-pong as well. Thus, we have:

$$R_i + D_{ij} \leq \bar{R}_i \tag{3.4}$$

Consequently:

$$D_{ij} \leq min(R_j - \bar{R}_j - R_{\Delta}, \bar{R}_i - R_i) \tag{3.5}$$

which is used to calculate the maximum allowable size of data transfer. The calculation is based on merely local information obtained through the aforementioned advertisement messages. EnviroStore then fetches a small chunk of data (i.e., a log item as defined in Section 3.3) from local storage, and, if its size is below the maximum allowable size, reliably transfers it to the redistribution target. Data chunks are transfered until the conditions specified by Inequality (3.1) and Inequality (3.2) are invalidated.

Besides flash overflow, an important factor that may cause data loss is depletion of node energy. When a node runs out of energy, the data stored at the node can still be recovered by collecting the deployed node. However, the node obviously ceases to observe the environment, losing all subsequent measurements. To avoid such data loss, besides monitoring remaining free storage, our algorithm also keeps track of the remaining node energy (by reading the current voltage level and converting it to energy based on battery characteristics). A node i should not invoke or accept data redistribution (which spends extra energy to send or receive data) unless its estimated energy lifetime is longer than its estimated storage lifetime. This leads to:

$$\frac{\Omega_i}{E_i} > \frac{R_i}{S_i} \tag{3.6}$$

where Ω_i is the remaining energy, E_i is the initial energy, and S_i is the initial storage size.

Cross-partition Data Redistribution

The in-network data redistribution scheme works well when all sensor nodes are connected into a single network. However, in some application scenarios like the GDI deployment [83], sensor nodes are naturally deployed into network islands that can not communicate with each other. Even for an initially connected sensor network, practical issues like the instability of wireless channels, hardware or software failures, or depleted batteries may separate the network into islands. In such circumstances, it becomes critical to offload data from overloaded network partitions (in terms of storage capacity) to either data sinks (if present) or underloaded partitions. This is accomplished via mobile data mules.

We consider two types of (authenticated) mules. The first always carries the data back to the base station. A node encountering such a mule can upload all its data to it. The second type is one whose mobility patterns are dictated by factors external to the storage system. The library example presented earlier is one such case. Such data mules can carry data to the base station if they happen to come in contact with it. They can also be used for data redistribution across network partitions.

To identify nodes as overloaded or underloaded for redistribution purposes, a data mule must have a notion of a global average storage use. Accurately calculating the global average is virtually impossible, considering that the sensor nodes in different partitions can not directly communicate. Instead, each data mule m remembers the free storage advertised by each visited node. It uses their average $\bar{R}$ as an approximation of the global average. It then computes its own advertised free storage value $\bar{R}_m$ as the weighted sum $\alpha\bar{R} + (1 - \alpha)R_m$, where R_m is the storage available on the mule itself. The parameter α of the mule is used to favor data redistribution versus upload. If α is close to 1, the mule favors redistribution to the neighborhood regardless of the storage available on the mule itself. If α is close to zero, it emphasizes upload, regardless of free storage available in the sensor network. While the mule remembers the R_i's of all encountered nodes, scalability is not a big concern since (i) data mules are more powerful than sensor nodes and (ii) keeping the storage information of one thousand nodes only takes several kilo-bytes of memory. Mules upload to a base station whenever possible. Observe that such an upload resets R_m to its maximum causing the mule to attract more data.

When mules have large storage or encounter the base frequently, to relieve the network storage, they should aggressively download data from the sensor nodes. To take this factor into account, a node i uses a weighted value R_iO_m (O_m is the occupancy ratio of the data mule m defined as the fraction of its local storage that is utilized) instead of its original remaining storage R_i to compare with $\bar{R}_m$. Obviously, the policy makes a mule download data more aggressively from sensor nodes when its occupancy is low (either because it has large unused storage or because it has offloaded most data to data sinks). Therefore, the cross-partition redistribution scheme can dynamically adapt itself to the differences in the size of storage space at mules as well as adapt itself to the visit frequency of data sinks if they exist.

Like sensor nodes, data mules also periodically advertise themselves by messages, but with a much higher frequency (e.g., once per second). Frequent mule advertisement is necessary for overloaded nodes to detect nearby mules as soon as possible in order to make the best use of data upload opportunities, as is shown in Figure 3.2. Frequent mule advertisement is feasible in terms of energy consumption since mules can be recharged frequently to obtain sufficient power.

At the same time, a mule should be able to detect nearby underloaded nodes to offload some of its data. The great difference between node advertisement frequency and mule advertisement frequency makes it much harder and slower for mules to detect nodes. To solve this problem, underloaded nodes, after receiving a mule advertisement, respond with a node advertisement to shorten the delay. These nodes use back-off timers (proportional to their current occupancy ratios) to suppress each other's node advertisement messages, as shown in Figure 3.2.

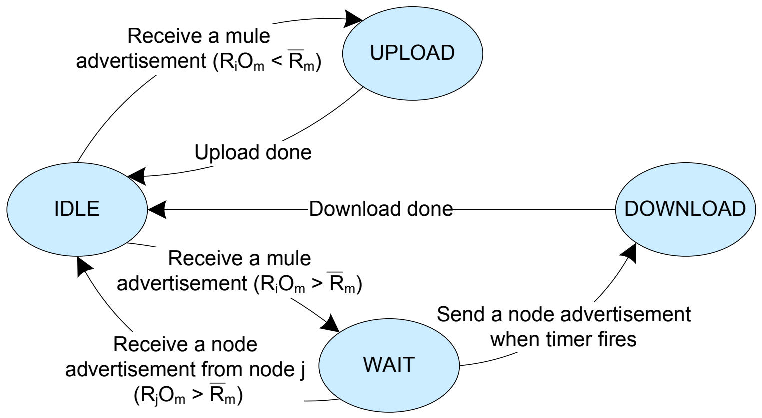

Receive a mule advertisement ($R_iO_m < \overline{R}_m$)
UPLOAD
Upload done
IDLE
Download done
DOWNLOAD
Receive a mule advertisement ($R_iO_m > \overline{R}_m$)
Send a node advertisement when timer fires
Receive a node advertisement from node j ($R_jO_m > \overline{R}_m$)
WAIT

Figure 3.2: State transition of a sensor node in cross-partition data redistribution

3.3 Implementation

We implemented EnviroStore using nesC in TinyOS. The implementation consists of three versions of code, encoding separately the subsystems run at sensor nodes, at data mules, and at data sinks. The overall system architecture for sensor nodes is depicted in Figure 3.3.

Application programmers use the service interface on the motes to submit data to be logged. Before data are written into local flash, such data are structured into the standard format of a *log item*, the minimum accessible data unit in EnviroStore whose data structure is depicted in Figure 4.4. Both writing and reading log items are functionalities supported by the *local log access* module. The *neighborhood monitor* module is responsible for sending advertisement messages within local neighborhoods. It also maintains a neighbor table to keep track of the storage status of each neighbor. At the same time, it is responsible for detecting mules via the reception of mule advertisement messages. Based on the conditions described in Section 3.2.2

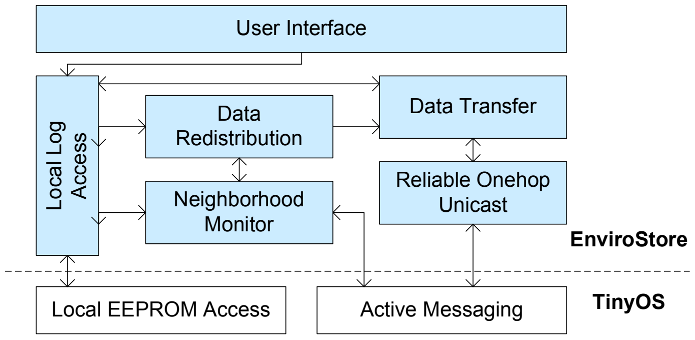


Figure 3.3: System architecture of EnviroStore

and 3.2.2, the *data redistribution* model determines whether the current node should offload data to it neighboring nodes or the detected mule, and, if so, it calculates the maximum amount of data transfer and signals the *data transfer* module to start data transfer towards a selected neighbor or the nearby mule. The *reliable one-hop unicast* module, as its name suggests, provides reliable unicast for nodes to transfer log items. To avoid data loss during transfer, the *data transfer* module never deletes a log item until the *reliable one-hop unicast* module acknowledges its reception.

The implementation for data mules has a similar set of modules except that the neighborhood monitor module records all the nodes a mule has met (in other works, dynamic neighbors of the mule) rather than a static set of neighboring nodes.

3.3.1 Local Storage Structure

The local storage space of nodes is organized into a circular buffer containing continuous log items (Figure 4.4). The *head* points to the next log item to be read or deleted, and *tail* pointer points to the next position to write a new log item. This simple data structure proves to be very suitable for current sensor network platforms. First, it meets all the requirements of EnviroStore since the system model suggests that random access to the logs is not required since we are not targeted for runtime data acquisition. Second, it consumes minimum code and data memory as this data structure organizes occupied space into a continuous data chunk, eliminating the need for any complex space management mechanisms like free space management or defragmentation. Third, it may prolong flash lifetime by balancing write access to different locations. Note that the endurance of the 512 KB serial flash on most current TinyOS platforms is only 10,000 erase/write cycles. The sequential write access to flash of the circular buffer structure guarantees that the number of writes to different flash blocks is almost perfectly balanced with a maximum difference of one.

As Figure 3.4 shows, each log item contains some metadata, including: (i) *log name*, the ID of the corresponding environmental event that serves as the source of the log item (more detailed explanation is give in Section 3.3.2); (ii) *log type*, used to denote the meaning as well as the structure of the logged data (e.g., temperature, noise level, target position, etc.) for later interpretation at data sinks; (iii) *source ID*, to indicate the node that originally generate this log item; (iv) *SN* and *count*, with the former used for ordering log items and the latter indicating how many logs of the same type that this item contains; (v) *log size*, the size of the logged data.

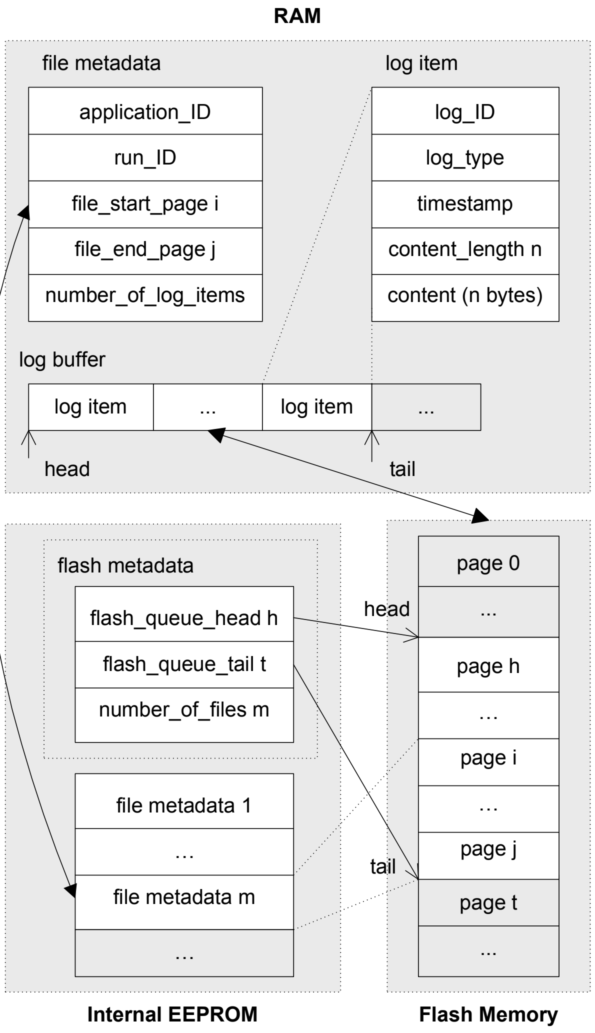
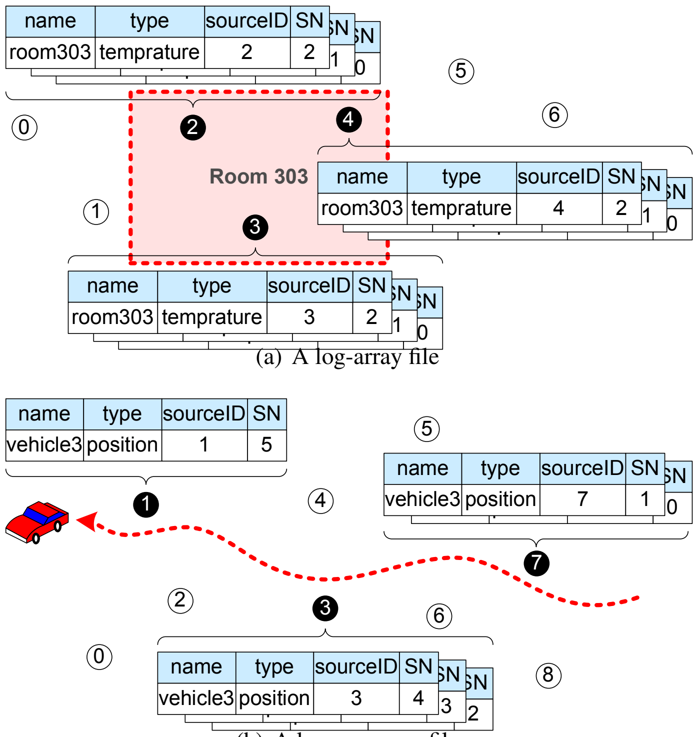


Figure 3.4: Local storage structure

Logically, the set of log items with the same name and type actually contains an attribute (e.g., temperature) or a set of attributes about a certain environmental event observed at different times and by different nodes. We call such a set a *log file*.

3.3.2 User Interface

EnviroStore supports two types of log files as Figure 3.5 depicts, and provides two nesC commands for them, respectively. The first type of log file is simultaneously written by different nodes, with each node generating a sequence of log items with continuous serial numbers. All the log items from multiple nodes form an array of log sequences. Therefore, we call this type of log file *log-array files*. Log-array files are useful for logging attributes of an environmental event that is independently monitored by multiple nodes, for example, to obtain the temporal and spacial distribution of the temperature in Room 303 as shown in Figure 3.5(a).



(a) A log-array file

(b) A log-sequence file

Figure 3.5: Examples of different types of log files

The other type, named *log-sequence files*, expects one writer at a time. Multiple nodes should coordinate with each other so that the next writer does not start before the previous one stops. Unique and continuous serial numbers must be used. This mode is developed for compatibility with EIP object abstractions. The service elects a unique leader node in the vicinity of the tracked entity and hands off leadership from node to node as the entity moves. The leader maintains a unique ID. This ID can be used as the log name to produce a distributed log-sequence file that stores the history of a target along its trajectory. An example is shown in Figure 3.5(b), where EnviroSuite associated an ID (`vehicle3`) to a vehicle, and elected nodes 7, 3 and 1 sequentially to log the vehicle's current state. The resulting log-sequence file, using `vehicle3` as its name, contains the trajectory of the vehicle.

Figure 3.6 lists the interface provided by EnviroStore. Users call the command `log()` to submit data to be logged. To simplify its usage, we do not expose `SN` to users by maintaining it inside EnviroStore. `SN` is initially set to 0, then automatically increases itself by one when a new log item with the same name and type is submitted. The event `logDone()` gives users feedbacks on whether the submitted item has been successfully written into flash or not.

```
interface LogIt {
  command result_t log(uint8_t name, uint8_t type, uint8_t size, uint8_t *data);
  event result_t logDone(uint8_t size, uint8_t *data, result_t success);
  command result_t continueLog(uint16_t nodeID, uint8_t name, uint8_t type);
  event result_t logContinued(uint16_t nodeID, uint8_t name, uint8_t type, result_t success);
}
```

Figure 3.6: User interface

The command `continueLog()` and the event `logContinued()` are used for creating log-sequence files. One node can request one of its neighbors (denoted by `nodeID`) to continue to log into the same log-sequence file by calling `continueLog()`, which actually sends `logName`, `logType` and current `SN` of the file to the neighbor. Upon the reception of this message, the designated neighbor sets the corresponding `SN` to be the received one, sends back an acknowledgement, and then continues to write into the file. When the sender successfully receives the acknowledgement, the event `logContinued()` is signaled.

3.4 Evaluation

This section presents a performance evaluation of EnviroStore. EnviroStore is implemented in nesC for TinyOS, which, after compilation, can be directly used in various hardware platforms supported by TinyOS. Instead of using real hardware, we use TOSSIM that provides a high fidelity simulation of TinyOS applications, precisely modeling the 40Kb network at the bit level and the CPU clock at a 4MHz granularity. Note that we did not provide a comparison between EnviroStore and other distributed storage systems due to the lack of appropriate systems to compare with. Most existing storage systems [18][66][99] are designed for connected operation, focusing more on indexing or query processing techniques to retrieve data at runtime. A storage system that shares a similar goal of storing long-term information is DIEMSIONS [24]. However, it utilizes lossy data compression techniques rather than data balancing (as in EnviroStore) to achieve the goal. Moreover, it is still under development and there is no implemented version we can compare against.

Figure 3.7 depicts the basic deployment configuration used throughout the evaluation. On a 80×80 ft^2 field, we deploy 36 nodes (circles labeled with node IDs) into four network partitions. Nodes in black (node 4 and 31) are data generators. They run an application that periodically creates input for EnviroStore. An unbalanced deployment configuration is used to stress EnviroStore. In the top-left and the bottom-right network partitions, only one node generates all the input, the others being silent. The other two partitions do not have input to further stress the redistribution algorithm.

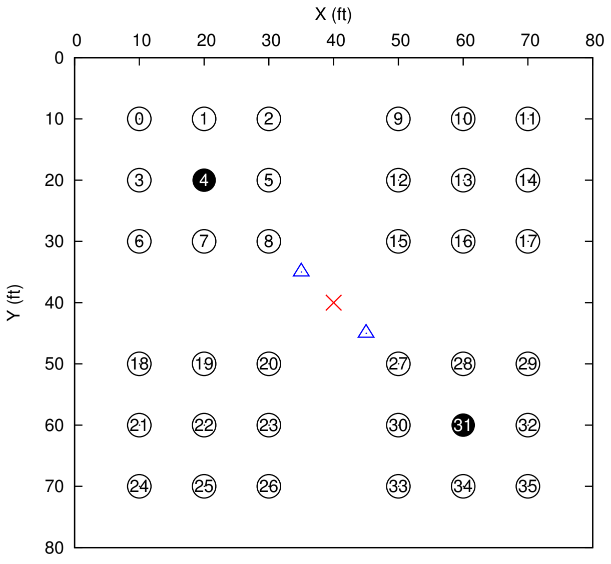


Figure 3.7: Basic deployment configuration

When present, a base station is placed at the position marked by $\times$, which is also the starting point of data mules. The movements of the mules follow a constraint random walk model, moving 5ft every second and turning a random angle between $-\frac{\pi}{6}$ and $\frac{\pi}{6}$. In some scenarios, extra nodes are deployed at the two positions marked by $\triangle$. The random walk serves our purpose well because EnviroStore has no knowledge about mobility of mules.

The final simulations of our experiments are very heavy-weight, especially when mules are used, mainly because of frequent channel update in TOSSIM caused by the mobility. For a network of 36 nodes, it takes

6-10 hours of wall-clock time on a Pentium4 1.7GHz machine with 1GM RAM to simulate 3600 seconds of virtual time. To accelerate the evaluation, we set storage capacity of the devices to be smaller than that of current hardware platforms. The storage of the node (S) is set to 16KB. The mules have a relatively larger storage of 64KB. Consequently, while the *absolute time* when the system encounters data loss will not match real platforms, relative performance of different algorithm will remain the same. Hence, inferences can be made about multiplicative improvement factors over a baseline.

Unless otherwise indicated, R_{TH}, $R_{gradient}$, and $R_{\triangle}$ are set to be $0.95S$, $0.05S$, and $0.01S$ (S is the total storage of a node), respectively. Recall that R_{TH} has to be big enough to accommodate bursts of input. We use a large percentage of the storage as R_{TH} since the total storage size is small. Next, we investigate a disconnected sensor network with and without partitions.

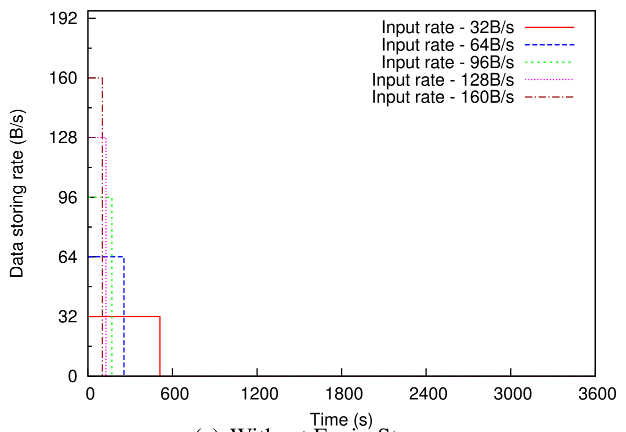
3.4.1 Scenario 1: Single Disconnected Sensor Network

In a single disconnected sensor network, the network is not partitioned. Also, neither mules nor a base station is present. For this scenario, we use only the top-left partition shown in Figure 3.7.

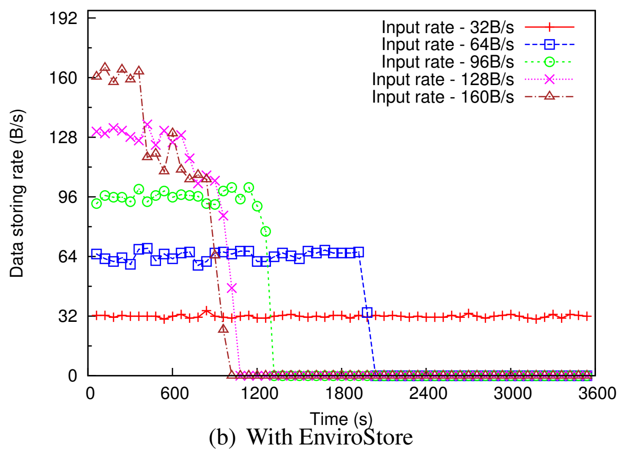
To illustrate how EnviroStore maximizes storage capacity via in-network data redistribution, Figure 3.8 compares the data storage rates (i.e., the amount of data stored by EnviroStore per second) for different input rates with and without EnviroStore. If storage is infinite on each node, the data storage rate always equals the input data rate. However, data loss caused by insufficient local storage makes the data storage rate drop below the input data rate. As can be seen in Figure 3.8, for all the input rates, applying EnviroStore significantly delays data loss. For example, with an input rate of 64B/s, the first appearance of data loss is delayed from time 256s to 1900s (i.e., more than 6 times).

Notice that for input rates lower than 96B/s, the corresponding data storage rates stay close to the input rates until they sharply drop to zero. In contrast, for input rates higher than 96B/s, the data storage rates decline *gradually*. The underlying reason is that when the input rate exceeds a node's communication bandwidth, new data arrives before the data redistribution algorithm converges to a balanced state. When the data generator (node 4) completely consumes its local storage, EnviroStore redistributes at the communication rate, which is below the input rate, yet above zero.

To investigate the effects of R_{TH} on the data storage rate and energy consumption, we fix the input rate to be 64B/s and use different values of R_{TH} ($0.9S$, $0.5S$, and $0.1S$). Using a smaller R_{TH} does not have



(a) Without EnviroStore



(b) With EnviroStore

Figure 3.8: Data storing rate at different time

an appreciable impact on the data storing rate. However, as expected, it does affect energy consumption, as shown in Figure 3.9 which plots the number of data messages sent per second for different values of R_{TH}. As can be seen, setting R_{TH} to $0.5S$ postpones extensive data transfer from 180s ($0.9S$) to 540s. Using $0.1S$ as R_{TH} further postpones it to 1200s. If the application input happens to stop at 1200s, setting R_{TH} to $0.1S$, comparing to $0.9S$, yields significant energy savings due to lazy offload.

3.4.2 Scenario 2: Partitioned Sensor Network with Data Mules

In order to analyze the effects of cross-partition redistribution, we deploy four network partitions consisting of 36 nodes as shown in Figure 3.7, as well as one mobile data mule. The input rates at node 4 and node 31 are set to be 64B/s and 32B/s, respectively. This setting provides the four partitions with three levels of input rates: 64B/s for the top-left one, 32B/s for the bottom-right one and 0B/s for the others. Great differences

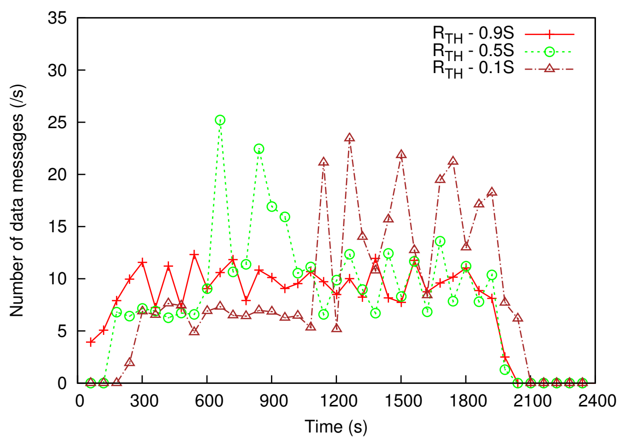


Figure 3.9: Comparison of data storing rate at different time

in input rates stress the distribution algorithm.

Figure 3.10 presents the data storing rate of different configurations. Ideally, the data storage rate should always equal to the input rate (96B/s). For the current configuration, without EnviroStore, the data storage rate drops from 96B/s to 32B/s after 256s when node 4's storage is exhausted, and to 0 after 512s when node 31's storage is exhausted. After applying in-network redistribution (without mules), data loss does not occur until after about 1900s. If we further invoke cross-partition redistribution by introducing a mule, the data storage rate stays at 96B/s until after 2400s. Overall, applying EnviroStore delays data loss by a factor larger than 8.

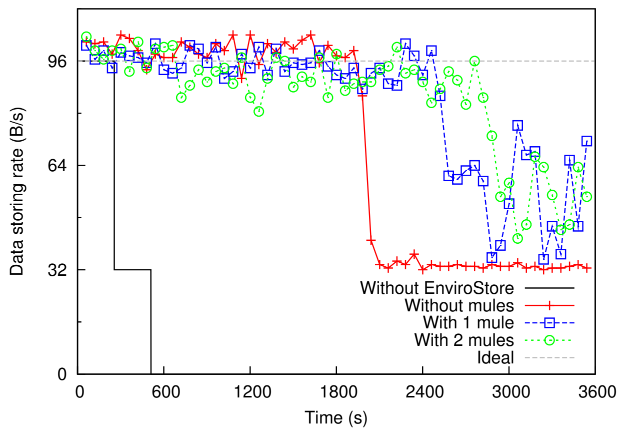


Figure 3.10: Comparison of data storing rate at different time

By carrying data from overloaded network partitions to underloaded partitions, the data mule delays the time that the storage of the most-loaded partition (the top-left one) gets exhausted. Figure 3.11 demonstrates this effect by showing the total stored data at node 4 over time. Obviously, in-network redistribution (without mules) reduces the storage consumption speed at node 4. Cross-partition redistribution (with a mule) further slows down the storage depletion.

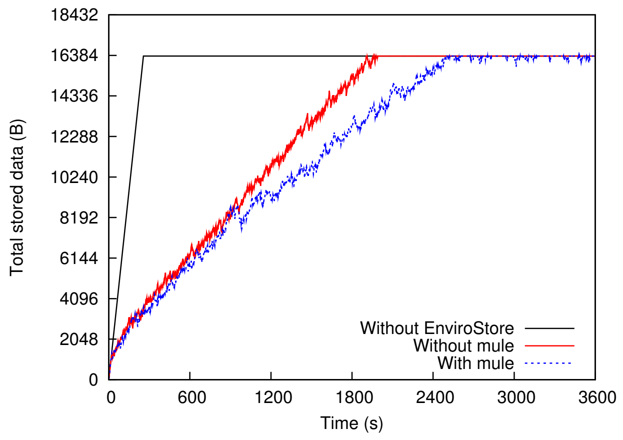
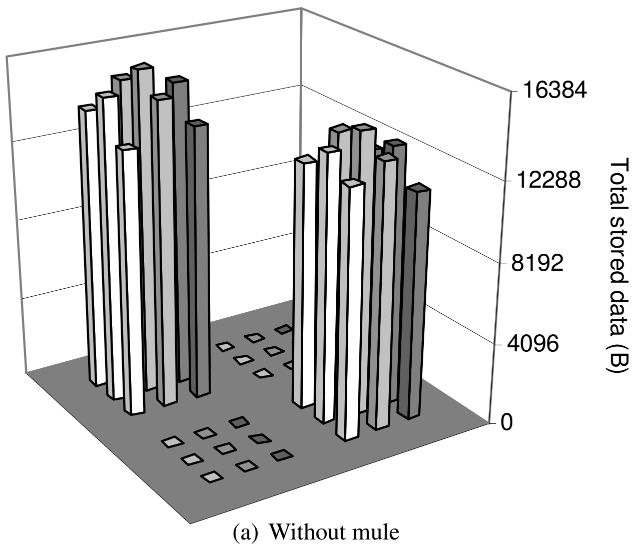


Figure 3.11: Comparison of total stored data of node 4 at different time

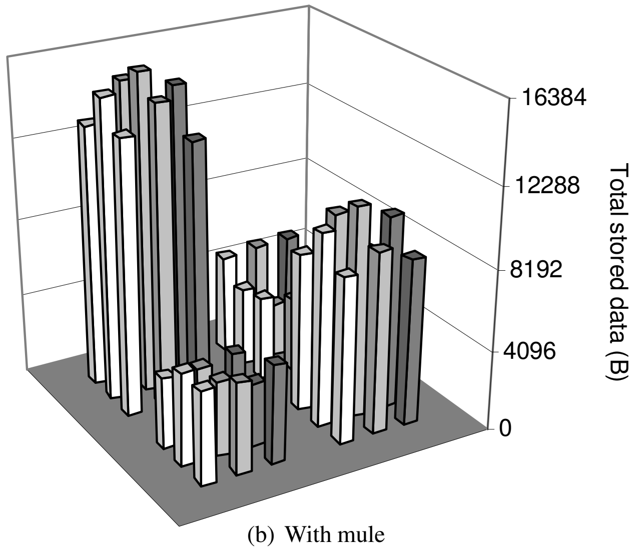
Figure 3.12 illustrates the distribution of total stored data among different nodes after 3600s of virtual time for deployment configurations without and with the mule. Obviously, the mule successfully moves data from overloaded network partitions to underloaded partitions and achieves a more balanced storage occupancy.

We also explore the energy overhead (in terms of messages) of redistributing data to maximize effective storage. The number of advertisement messages and data messages sent per unit time by the sensor nodes for both configurations with and without the mule are plotted in Figure 3.13. As shown, the total number of messages sent per second for the whole network is always below 36, in other words, below 1 per node, which is acceptable for sensor networks. For the configuration without a mule, after around 1900s, the number of data messages per second drops to 5 abruptly and the number of advertisement messages decreases as well. This is when the top-left network partition gets exhausted and stops accepting input as well as data redistribution, which is consistent with what we observe in Figure 3.10. We do not see such a sudden drop in the configuration with the mule because (i) prior to time 2400s the top-left partition is not full, and (ii) after time 2400s the mule is still actively communicating with the nodes to do cross-network redistribution.

Finally, we add a base station, marked by $\times$, and two extra nodes, marked by $\triangle$, in Figure 3.7). They ensure connectivity between the sensor nodes and the base station. Figure 3.14 shows the data storage rate of the base station over time. Recall that we allow certain storage imbalance between nodes by using $R_{gradient}$



(a) Without mule



(b) With mule

Figure 3.12: Distribution of total stored data after one hour

of $0.05S$. Therefore, the data storage rate of the base station stays zero for a certain duration, then increases gradually.

In the next experiment, we remove the two nodes marked by $\triangle$ in Figure 3.7 to disconnect the network partitions from the base station and add one or two data mules to see how they help with data upload. Figure 3.15 depicts the amount of accumulated data at the base station over time. Having more mules increases the rate of uploading data to the base station, as shown in Figure 3.15.

Adding extra mules obviously increases upload rate if comparing the curve of two mules with the curve of one mule. However, the observed upload rate is still low. We are expecting much higher upload rate in

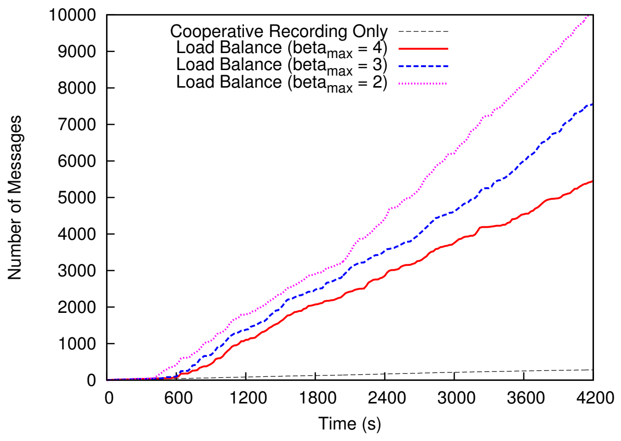


Figure 3.13: Comparison of number of messages per second at different time

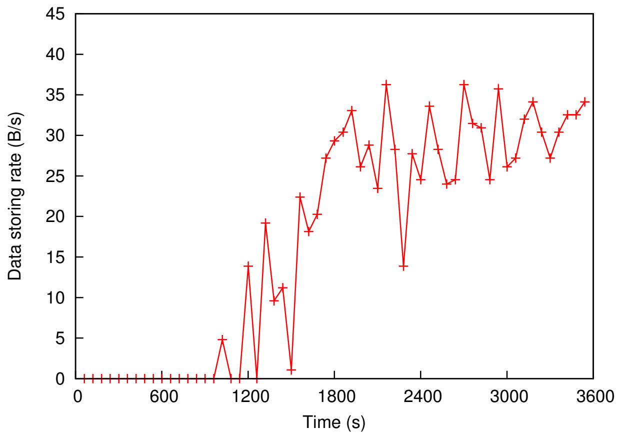


Figure 3.14: Data storing rate of the base station over time

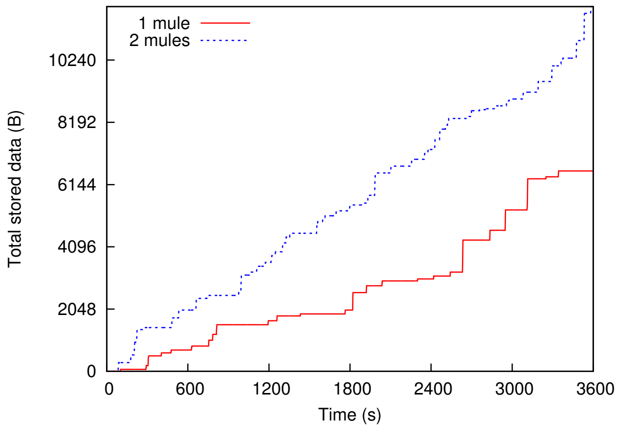


Figure 3.15: Total stored data of the base station over time

reality since (i) current hardware platforms usually have higher bandwidth (e.g., 250Kbps radios on MicaZ motes) than the simulated radio (40Kbps), (ii) to communicate with base stations, mules should be able to use an alternative high-bandwidth radio different from the one it uses to talk with sensor nodes, and (iii) using more mules also help increasing the upload rate.

3.4.3 Concluding Remarks

The evaluation results above demonstrate the effectiveness of EnviroStore in various application scenarios. Many environmental monitoring applications deploy single disconnected sensor networks to collect data. Nodes are collected after the experiment to download data from nonvolatile storage to do complex off-line analysis. For this category of deployments, EnviroStore is able to significantly reduce data loss by fully leverage network storage capacity as shown in Figure 3.8.

For other applications where the sensor network is naturally partitioned into several islands, EnviroStore can further maximize network storage by opportunistically offloading data from overloaded network partitions to underloaded ones via data mules. For our experiment configurations, introducing one mule postpones data loss onset by over 25%.

For the third category of applications whose deployments contain a base station connected to the nodes, EnviroStore can automate data upload towards without relying on multi-hop routing algorithms as is proved by Figure 3.14. If mules are deployed, the data uploading even does not require direct paths between sensor nodes and sinks, as Figure 3.15 shows.

In summary, the evaluation results demonstrate that EnviroStore is applicable to a wide set of application scenarios. It effectively maximizes the network storage capacity through in-network and cross-partition data redistribution.

3.5 Conclusions

In this chapter, we present EnviroStore, a cooperative storage system that maximizes network storage capacity in the presence of disconnected operation in wireless sensor networks using in-network and cross-partition data redistribution mechanisms. Our evaluation study validates that EnviroStore can (i) effectively utilize the network storage capacity of disconnected sensor networks to accommodate the most sensory data, and (ii) opportunistically offload data from overloaded network partitions to underloaded partitions via mules.

Using EnviroStore, EIP programmers can easily store the aggregate states of objects in the network for later retrieval for disconnected application scenarios. EnviroStore is also applicable to other programming models due to its generic interfaces. The system model based upon which we design EnviroStore is flexible enough to allow a wide set of applications to take the best advantage of EnviroStore.

Chapter 4

EnviroLog: The Replay Tool

4.1 Introduction

With the aid of intuitive programming abstractions and support middleware services of the EnviroSuite framework, programmers can design and implement environmental tracking and monitoring applications of sensor network in a more cost-efficient way. However, the developed applications are not ready for deployment yet. As sensor network systems are usually embedded and expected to work continuously for months or years, thorough testing and evaluation are absolutely necessary. Embedded devices usually have very limited or even no display, which makes sensor network debugging a more than challenging task. What further aggravates the situation is the fact that final system testing and evaluation have to be done under realistic environments, for sensor systems interact intensively with the external environment that is hard to simulate. For instance, to statistically measure the performance of a tracking system, targets of interests such as vehicles have to go through the field hundreds of times. To reduce such in-field testing cost, we present in this chapter a testing tool that aids in-field debugging and trouble-shooting by enabling repeatability of the external environment, called *EnviroLog*.

As demonstrated later, EnviroLog is quite useful in testing and evaluating environmental tracking and monitoring applications developed using EnviroSuite. However, most event-driven systems, not limited to those developed using EnviroSuite, should benefit from EnviroLog.

More broadly, with the increase in maturity of sensor networks research and with recent solutions to several practical systems and deployment problems, sensor network applications are entering the real world. Representative experiences of such evolution are documented in recent literature such as military surveillance [37, 80], habitat monitoring [67, 59], and environmental monitoring [98, 6], just to name a few. Quite different from controlled lab settings, physical environments introduce a high degree of uncertainty that makes it hard to conduct reproducible experiments. Consequently it is hard for researchers to obtain statistically consistent empirical results. With the growing number of applications developed and deployed, there is an increasing need for tools and services to assist with system evaluation and debugging, as well as with performance tuning of applications in outdoor environments.

To address this issue, we propose EnviroLog, a tool to improve repeatability of experimental testing of distributed event-driven sensor network applications. Unlike time-driven applications, such as periodic sampling of environmental conditions, the state of event-driven systems can change depending on the particular sequence of events received and their timing. Matters such as tuning protocol parameters for particular event scenarios or comparing performance of different protocols typically require the same distributed event traces to be replayed (e.g., to ensure a common basis for comparison). To address this requirement, we provide an event recording and replay service that can capture and reproduce distributed events on demand. The service provides the abstraction of a completely repeatable environment as observed by the sensory subsystem for the sake of experimental testing. The service is especially valuable in the study of rare, unsafe, or hard-to-reproduce events such as the motion of tracked animals through a sensor field.

Our service is geared for the final stages of testing, typically performed in-field, where the effects of environmental realities can be studied. Early testing can use simulators that may be good for initial debugging since they allow fully controlled and repeatable experiments to be conducted. However, simulators are notoriously inaccurate when it comes to sensor network applications, since: (i) certain practical issues (e.g., the distribution of radio irregularity) are not adequately captured in most available simulators, resulting in large discrepancies between simulation results and empirical measurements, and (ii) simulators do not faithfully mimic environmental event signatures which affects the performance of sensor network applications. The problem is especially severe in event-driven architectures, where application behavior is more sensitive to the sequence, timing, and parameters of events received.

In addition to improving the repeatability of field testing, our service significantly reduces experimentation cost. In the absence of a recording and replay service, the investigators would have to either physically reproduce or passively wait for environmental events of interest, which entails additional costs. For example, the authors of [37] have developed a surveillance system that tracks persons and vehicles in the field. The need for walking or driving through the field hundreds of times while tuning an array of protocol parameters has proved to be a major practical impediment imposing a significant limitation on the rate at which experiments could be conducted in practice. EnviroLog, the asynchronous event recording and replay service described in this chapter, provides a comprehensive solution to this problem.

Most sensor platforms employ flash memory for persistent storage. For example, Mica, Mica2, Mica2-Dot [41, 42] and XSM [74, 21] hardware platforms incorporate 128 KB internal flash for code storage and 512 KB external flash for other usage. EnviroLog logs environmental events into such persistent storage devices. Later on, in replay mode, EnviroLog replaces environmental inputs with retrieved logs and re-issues the logged events in their original time sequence as asynchronous inputs. In addition to environmental events, EnviroLog can also log system runtime status for future analysis by recording selected variable values at runtime as specified by the programmer using simple annotations.

EnviroLog has two unique features. First, EnviroLog can operate at any layer of an application. In other words, events recorded and replayed by EnviroLog are not limited to direct reflection of environmental events such as raw sensory readings. They can be any system-level events. This characteristic of EnviroLog enables the debugging or tuning of any specific layer using controlled and repeatable inputs from lower layers in an event-driven system. Second, EnviroLog provided a very friendly user interface. Users only need to add simple annotations before events or variables to be logged. Applications with EnviroLog annotations can be compiled either into production code that ignores all EnviroLog annotations or into development code that allows on-demand recording and replay.

The potential uses of EnviroLog include (i) in-field debugging and performance tuning of specific parameters of an application, (ii) collecting statistical results from a large number of repeated experiments, and (iii) generating traces for mixed simulation environments that accept experimental measurements as inputs.

EnviroLog subjects to certain limitations in terms of scope of applicability. First, communication properties remain stochastic when EnviroLog is used to merely capture the logical effects of sensor subsystems at individual nodes. In such case, system output during replay is expected to be different from during recording due to dynamics in communication conditions. This limitation is actually beneficial in certain sense since it achieves the separation of the effects of communication non-determinism and the effects of environmental non-repeatability in the study of sensor network protocols. Second, a critical assumption of EnviroLog is that the lower layers of the application (i.e., the layers below EnviroLog) always create the same set of events for the same environment. This assumption does not hold in all applications. For example, certain applications are integrated with power management schemes which may turn off nodes to save energy. If the sleep schedule of a specific node changes between recording and replay, the set of logical events should change as well even for the same environment. It may occur that data to be replayed does not exist since the node was asleep during the recording stage. EnviroLog is not applicable to such systems.

The remainder of the chapter is organized as follows. Section 4.2 describes the design goals and system architecture for EnviroLog. Section 4.3 describes the implementation details. Section 4.4 evaluates EnviroLog, using XSM platforms, through a series of in-field experiments based on several sample applications provided by TinyOS [40] and a surveillance system [37] built upon TinyOS. Section 4.5 concludes the chapter.

4.2 Design

The major goal of EnviroLog is to enable experimental repeatability by recording and replaying environmental inputs. From a program's view, such inputs are essentially data streams that are transformed by each module of the program until they reach the modules at the topmost layer and become outputs. If we log the data stream generated by a module during the occurrence of an environmental event, and regenerate the same data stream later, from the perspective of the modules that consume the data, the environmental event is repeating itself.

EnviroLog makes the assumption that data is transferred from one module to another module only through function calls and their parameters. This assumption conforms to the event-driven sensor acquisition convention of TinyOS, where sensory data is usually obtained through asynchronous events (defined as a type of function calls in TinyOS) with parameters containing the data. Based on this assumption, the desired data stream generated by a module can be recorded by logging all its issued function calls and their parameters. That founds the main idea that EnviroLog is built upon.

Users of EnviroLog, through user interfaces, designate the modules that provide the data stream. These modules are called *log modules*. The modules that directly or indirectly consume the logged data stream are called *target modules*. As depicted in Figure 4.1, during the *record stage*, EnviroLog logs all function calls issued by the log modules into persistent storage devices such as a flash. During the *replay stage*, log modules are disabled. Instead, EnviroLog issues the previously recorded function calls at the right time and in the right sequence as recorded. Based on this main idea, the following subsections discuss the design of EnviroLog in more detail.

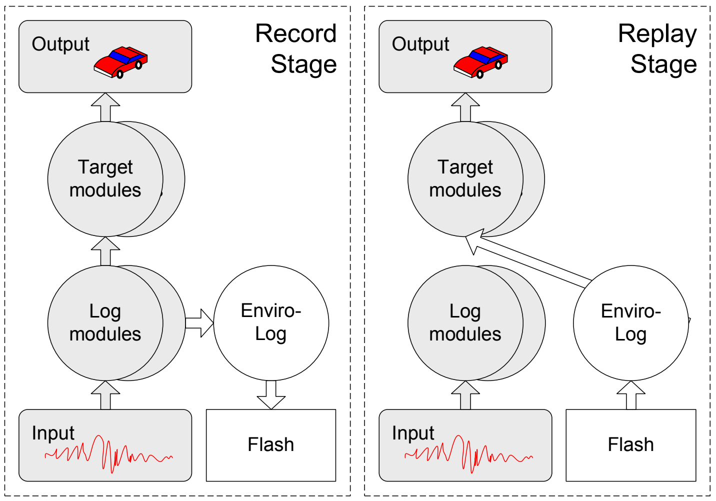


Figure 4.1: Main idea of EnviroLog

4.2.1 Design Goals

The design goals of EnviroLog are:

- **Effectiveness:** Effectiveness of EnviroLog is measured by its ability to perform: (i) accurate event record and event replay, and (ii) reliable runtime status record and retrieve.
- **Efficiency:** In sensor platforms, resource constraints (on both CPU and memory) are significant which makes it critical to use resources efficiently. EnviroLog is designed with efficiency in mind so that it can be applied to complex applications that consume a significant fraction of available resources.
- **Simplicity:** As a tool aimed to simplify the life of sensor network programmers, EnviroLog must be easy to use. In other words, simple user interfaces should be provided. Experiences tell us that simple tools tend to gain more popularity and persist longer while more complex tools sometimes run into usability barriers and become deserted.
- **Flexibility:** There are always tradeoffs between performance and overhead. Since EnviroLog cannot achieve optimal performance and minimum overhead at the same time, it should allow users to flexibly select their specific performance requirements (e.g., high logging throughput) at the expense of incurring a corresponding overhead. EnviroLog should also be flexible enough to allow users to record and replay any application behavior not limited to direct inputs of environmental events.

The above design goals provide guidelines throughout the design and implementation of EnviroLog.

4.2.2 System Architecture

Figure 4.2 illustrates the architecture of EnviroLog. Users indicate the set of events and variables to be recorded by inserting special annotations, called *EnviroLog annotations*, into applications. EnviroLog annotations are essentially the user interface of EnviroLog. Annotated applications are then processed by a *preprocessor* that translates EnviroLog annotations into real code, and, based on the annotations, integrates only the necessary modules from a *code repository* into the applications. The preprocessor and the code repository constitute *EnviroLog services*. As a result of the processing, application code is given the ability to record and replay specified events and to record and retrieve specified variable values that represent runtime status.

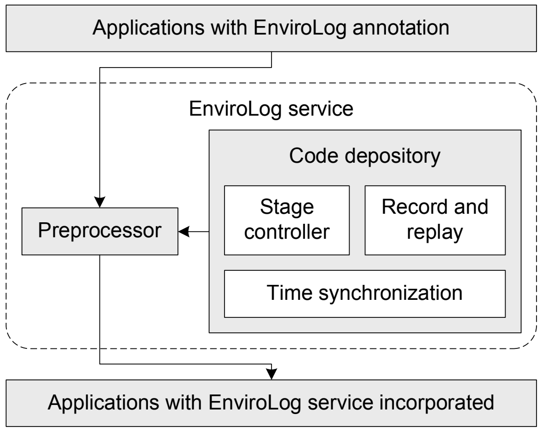


Figure 4.2: System architecture of EnviroLog

4.2.3 User Interface

Our design goal of simplicity calls for an easy user interface through which users are able to express the desired functionality of EnviroLog in a simple and intuitive manner.

The user specifies two main issues: asynchronous environmental event record and replay, and runtime status record and retrieve. The user interface design for the latter is easier since runtime status can be interpreted as values of variables at runtime. Such variables are simply annotated for logging. The former one is more complicated. As stated in the beginning of this section, EnviroLog supports environmental event record and replay by logging and regenerating data streams originating from environmental events. Since these data streams are transferred between modules through function calls and their parameters in most embedded systems such as TinyOS, we can log the behavior of a module by recording all issued function

Table 4.1: EnviroLog annotation list

Purpose	Annotation	Usage	Functionality
For event record and replay	/*LOG_MODULE*/	Insert the annotation in the beginning of the implementation of a module	To record all function calls issued in the module for future replay
	/*LOG_FUNCTION*/	Insert the annotation before a clause that makes a function call	To record the function call for future replay
For system status record and retrieve	/*LOG_VARIABLE: variable_name*/	Insert the annotation with specified variable name at a position within scope of the variable	To record current value of the variable for future retrieve

calls. Therefore, the only action an EnviroLog user is required to take is to specify a set of data-providing modules whose output data streams (issued function calls) are to be logged. The preprocessor takes the responsibility of enumerating function calls within the annotated modules and using APIs to log and replay each of them. To be more flexible, instead of specifying entire modules, advanced users are allowed to specify the exact function calls to be logged within a module.

Different from most services that usually provide function call APIs, EnviroLog provides EnviroLog annotations as the user interface. Table 4.1 lists the basic set of EnviroLog annotations.

The special characteristics of this user interface are:

1) EnviroLog annotations are simple to use. Users only need to insert annotations before function calls without worrying about details such as how these function calls are recorded, what data structures are used, and how and when to re-execute them when replaying. The preprocessor takes the responsibility to automatically generate code and integrate modules from the code depository to handle these details.

2) EnviroLog annotations take the form of comments that are ignored by the regular language compiler. This allows users to freely switch between original applications without EnviroLog integrated and EnviroLog-augmented applications. Annotated applications, when directly compiled, generate executables that do not include EnviroLog support. Alternatively, if they are processed by the EnviroLog preprocessor before compiling, the resulting executables are able to record and replay/retrieve the specified set of function calls and variable values based on user annotations.

To log environmental events, one option is to log the inputs of sensor drivers. However, users may focus on the evaluation and tuning of a specific layer higher than the layer of sensor drivers, thus requiring repeatable inputs to this layer. To fulfill such requirements, EnviroLog allows users to use EnviroLog annotations at any layer to enable repeatable input to that specific layer without interference from lower layers.

Though EnviroLog is geared for recording and replaying events of interest, it is also possible to choose whether to record the radio channel conditions using the user interface. If function calls to send messages or to cause the sending of messages are logged, the channel conditions are not captured and might be different between record and replay, depending mostly on the environment. Alternatively, if the programmers choose to log function calls that handle the reception of messages, the delivery of messages and their sequence should be the same between record and replay, no matter how the radio conditions change in reality. However, repeating the channel conditions is not always desired since in most experimental scenarios it is valuable to investigate how variances in channel conditions affect system behavior given the same sensory inputs.

4.2.4 Preprocessor

The simple user interface is supported by the preprocessor, which takes applications with EnviroLog annotations as input and outputs applications with EnviroLog code incorporated. The functionality of the preprocessor includes:

- Enumerating function calls and variables to be logged and assigning unique IDs, called *log IDs*, to them;
- Translating EnviroLog annotations into code that performs three functions. It defines the data structures for function parameters. It uses APIs provided by the record and replay module to record log IDs together with function parameters or variable values at the record stage. Finally, it re-executes at the replay stage the function calls upon the reception of logs from the record and replay module;
- Selecting necessary modules from the code repository and integrating them into applications.

robustness, logged data must be consistent with the current application during replay. Replaying data to the wrong application is not meaningful. A simple approach to ensure consistency is to use an ID, called the *application family ID* (or *application ID* for short) to denote the application tuned or the class of applications compared. The ID is logged as metadata at the record stage and is verified before replay. This ID is either specified by the application programmer who has the knowledge of which applications belong in the same family, or, if not specified, automatically created by the preprocessor by hashing the set of logged function calls into an application ID. The former solution allows for more flexibility while leaving the responsibility of ensuring consistency to the programmer. In the latter case, the same ID is generated as long as users don't change the set of function calls to be logged. This approach ensures consistency between logs and applications, making application IDs transparent to users since they are automatically handled by the preprocessor. Note that, both solutions enable repeatable environmental input to different application versions. Both the log modules and the target modules can be different between record and replay stage as long as data interfaces (in other words, the set of logged function calls) between them are the same.

Another challenge is to ensure *complete and consistent replay*, which means that:

- System outputs should be exactly the same during the record stage and replay stage, as long as target modules are not changed.

- Changes to target modules for purposes such as performance tuning should not affect data streams output by log modules. If the target modules can affect the behavior of log modules, EnviroLog design may be unrealizable. For example, if power management services can dynamically select a subset of nodes and turn them off, a situation can arise where a node is turned off during the record stage but turned on during the replay. It is obviously impossible to decide on the correct value to be replayed since none was recorded. Another example is when a different (e.g., faster) sensor sampling rate is set by target modules during replay. Since data was recorded at a different rate, the information to be replayed is not available in the log. Both of the aforementioned cases are hard to accommodate, and are therefore not allowed.

With the prior guarantee that the two special cases above don't exist in a given application, the preprocessor can provide some consistency checks. To enable these checks, in addition to the set of log modules (L), users are required to specify the set of modules (I) that directly interact with the environment (e.g., senor drivers) and the set of modules (O) that provide system outputs (e.g., modules reporting final decisions to base stations). If one module u issues one or more function calls to another module v, we denote this relationship as $u \rightarrow v$. To formalize the initial check procedure, we further define that i represents system inputs, o represents system outputs and M represents the set of all application modules. Based on user inputs (L, I, O) and definitions (i, o, M), the preprocessor abstracts the application into a directed graph $G = (V, E)$, where:

$$\begin{aligned} V &= \{u \mid u \in M\} \cup \{i\} \cup \{o\} \\ E &= \{(u, v) \mid (u, v \in M \wedge u \rightarrow v) \vee (u = i \wedge v \in I) \\ &\quad \vee (u \in O \wedge v = o)\} \end{aligned} \tag{4.1}$$

Given the set of log modules (L) specified by the user and the calling graph G, the consistency check is done first by removing all edges that originate from vertices representing these log modules from the graph, which forms a new graph $G' = (V', E')$, where:

$$\begin{aligned} V' &= V \\ E' &= E - \{(u, v) \mid u \in L\} \end{aligned} \tag{4.2}$$

If G' doesn't contain a directed path from i to o, the log module set is guaranteed to provide complete and consistent replay; otherwise, it may not be true.

Figure 4.3 gives an example to show more concretely the consistency check algorithm. Assume the application contains four modules, M_1 through M_4. Module M_1 directly consumes environmental inputs and issues function calls to M_2 and M_3. Both M_2 and M_3 further issue function calls to M_4. Finally, M_4 produces system outputs. The preprocessor builds the calling graph between application modules, adds output o and input i as virtual modules, connects input i to modules that directly consume environmental inputs (M_1), and connects modules producing system outputs (M_4) to output o. The result is the directed graph shown on the left rectangle of Figure 4.3. According to the algorithm, for each logged module, we then remove its outgoing edges from the graph. If the resulted graph doesn't contain any directed path from input i to output o, complete and consistent replay is guaranteed. As shown in the middle rectangle, logging of M_1 or logging of M_2 and M_3 ensures complete and consistent replay since no path exists from input i to output o. However, if only M_2 is logged, as shown in the right rectangle, a directed path traversing M_1, M_3 and M_4 exists between input i and output o, which fails in the consistency check and warns against a potentially incomplete or inconsistent replay.

Note that passing the initial checks is a sufficient, but not necessary condition for a complete and consistent replay guarantee, since not all function calls are issued due to environmental events. Thus, a directed path from i to o does not necessarily indicate that system outputs will indeed be affected by environmental

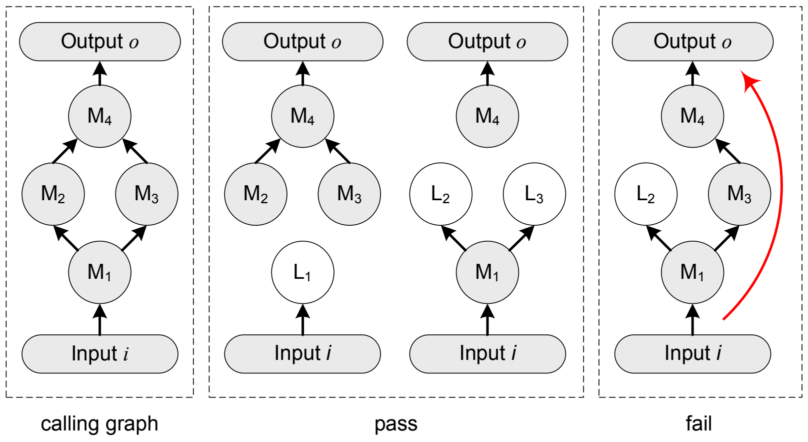


Figure 4.3: Examples of consistency checks

inputs.

4.2.5 Stage Controller

EnviroLog needs runtime facilities to control (during execution) when to record environmental events or system status and when to replay events as well as to retrieve recorded system status. For this purpose, we provide the *stage controller* to interact with users during runtime, employing a client/server architecture. The server node, connected to a PC, receives *commands* from users through a command-line interface or GUI and disseminates them to client nodes. Upon the reception of commands, client nodes execute corresponding code for commands immediately or at a requested future time.

Each command must include a *command name* and a *stage name*. Command names include *start*, *stop*, *pause* and *continue*. Stage names can be *record*, *replay* or *retrieve*. The time period between the start of a stage and the stop of that stage is called a *run*. Other optional parameters in a command include a *replay speed* for the replay stage to request that logged events be replayed n times faster or slower than their original rate. For a retrieve stage, variable names and/or node IDs can be specified to denote the name of a variable whose recorded values and timestamps in the specified node are to be retrieved. Any stage can also specify a *run ID* to select the logs for the particular run the command will operate on.

To provide accurate replay for applications involving multiple nodes, one critical factor is that they have to be synchronized. In other words, these nodes have to execute the same command at the same time. If the same command is always delivered to all nodes at the same time, we can simply program nodes to execute the command right after its reception without worrying about synchronization. However, the assumption doesn't hold for multi-hop applications or single-hop applications with lossy links. We solve the problem by proposing *two-phase command execution*, which makes use of time synchronization and system-wide broadcast. When issuing commands, users are required to provide a future time as a command parameter, which specifies when the command is to be executed. Then, in the first phase, the command and synchronization beacons originated from the server node are propagated across the entire network to synchronize clocks of client nodes as well as to broadcast the command. When the specified future time comes, client nodes enter the second phase simultaneously to execute the command. Two-phase command execution is costly because of its time synchronization service and repeated system-wide broadcast. To be flexible, advanced users are allowed to configure the stage controller into its lightweight single-hop version as well as a version with two-phase command execution support.

Researchers on time synchronization [14] for current hardware platforms have observed a significant variance in clock frequency due to the instability of the used crystals. Although mechanisms like linear regression are able to compensate for clock drifts in the short term (e.g., within 30 seconds), periodic re-synchronization through messages is inevitable for long-term experiments to keep the error to the microsecond range. As a result, long experiments tend to introduce more inaccuracy if using two-phase command execution. In such a case, an alternative solution would be to keep the time synchronization service on throughout the two phases assuming that re-synchronization messages do not alter the behavior of the applications.

4.2.6 Record and Replay

The record and replay service is the core component of EnviroLog. It responds to stage control commands to switch between different stages, logs data into flash during recording, reads from flash the logged function calls to re-issue them in their original time sequence during replay, and reads from flash the logged variable values during status retrieving. Besides maintaining logs of function calls and variable values, it also maintains metadata such as application ID, run ID, and run length during recording, which are to be verified when replaying. The service also supports the replay of events at a speed different from recorded one, which can be used to emulate extremely fast or slow targets that are hard to generate physically.

Queue-based File System

If metadata and logs of one run are viewed as one file, we can easily design the service based on existing file systems such as Matchbox [26] and ELF [17]. To be comprehensive, these file systems usually support various file operations such as open, close, read, write, and append, consuming a lot of code as well as data memory. Our design goal of efficiency calls for a simpler solution. Hence, we propose a *queue-based file system*, where files are organized into one queue. At any point in time, only the file at the tail of the queue is writable and new data is always appended to this file. Only the file at the head of the queue can be deleted. It differs from typical file systems in that (i) each file occupies a continuous storage space, and (ii) the gap between two successive files is always smaller than one page. The queue-based file system brings about several benefits:

- It realizes the special characteristics of the logging behavior in EnviroLog: logs are sequentially written into flash, and oldest logs are usually most undesired.
- It exhibits low resource consumption. This file system only supports a minimum set of operations (file creation, sequential write, sequential read, file deletion) that is necessary in EnviroLog. The queue-based design eliminates the need for complicated storage space management such as free page maintenance and flash defragmentation. Hence, it consumes minimum code and data memory.
- It prolongs lifetime of flash memory by balancing writes to different pages. Each flash page has a write limit of about 10,000 times. In the queue-based file system, the sequential write access to flash pages ensures that the number of writes to different pages differs at most by 1.

Distribution of Data Structures

Although the ultimate storage space for logs is flash, during runtime, multiple memory levels are employed to improve efficiency and reliability. Figure 4.4 depicts the distribution of data structures in RAM, internal EEPROM and external flash.

Because flash access is relatively slow (e.g., for flash AT45DB041B used in MICA motes, erasing a page takes up to 8ms and writing a page takes up to 14ms), the service employs a buffer in RAM, called *log buffer*, to temporarily store logs before committing them into flash. *Log items* constructed for function calls or variable values to be recorded are queued in this log buffer. Each log item consists of a log ID,

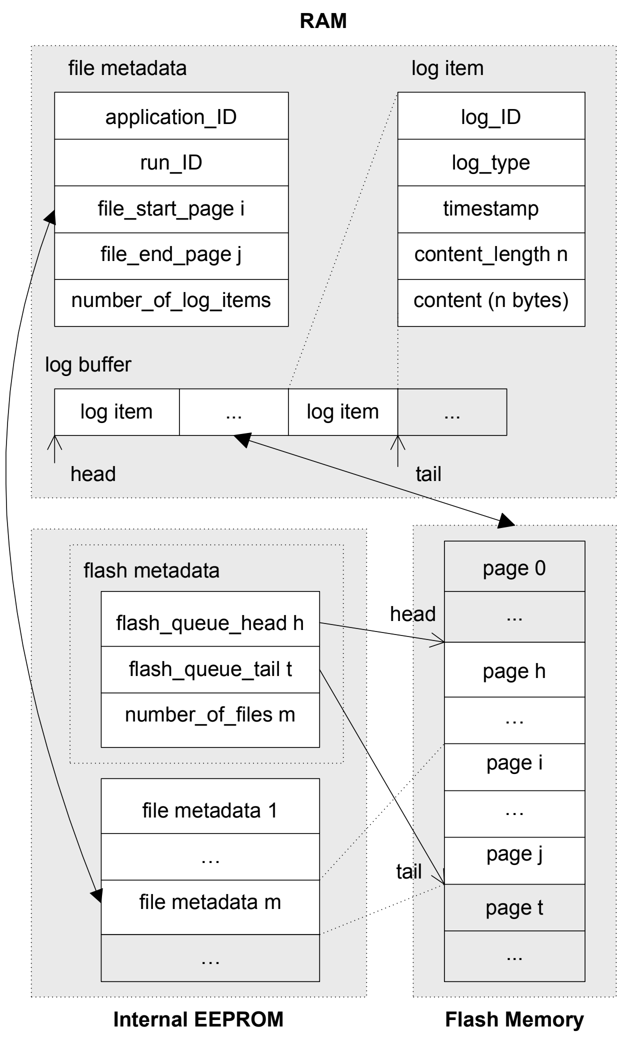


Figure 4.4: Data distribution in different memory levels

a log type (`event` or `status`), a timestamp, log content (parameters of function calls or values of variables) and content length. We note that buffering can not only increase throughput of flash access, but also accommodate temporary data bursts typical in event detection or tracking applications. Another benefit of buffering is to support potential data compression. Many log items can be compressed before being written into flash to conserve space, including those containing the same content but different timestamps and those containing timestamps with fixed differences.

All log items of a run constitute a *file*, which is usually stored in flash. Access to a file is usually sequential, either reading from the beginning to the end or writing from the beginning to the end. Besides files, the metadata of files (*file metadata* for short) and the metadata of the queue (*flash metadata* for short) also require permanent storage. Different from access to files, access to metadata is much more frequent, that is why we store such data into the internal EEPROM of MICA motes which is smaller (4KB) but with a longer endurance (100,000 write/erase cycles) compared with the flash memory (512KB, 10,000 write/erase cycles). For other sensor devices without EEPROM, EnviroLog can be adapted to reserve several pages in flash for metadata storage. These pages are expected to be worn out earlier than other pages, which makes the usable flash size smaller.

Workflow of Different Stages

This section explains the execution flow of the record and replay service during different stages in more detail.

At the beginning of a record stage, upon the reception of a `start record` command, the service reacts by: (i) remembering the current time as the reference time, (ii) loading flash metadata into RAM, and (iii) constructing the metadata of a new file. During the record stage, whenever the application requests to record a function or a variable value, the service constructs a corresponding log item and enqueues it into the log buffer. The log item contains a relative timestamp, which is calculated by subtracting the reference time from the current time. If more than half of the buffer is filled up, all log items in the buffer are transferred into flash, which empties the buffer completely. An alternative choice would be to write everything in the buffer into flash whenever a new log item arrives. We decide on the former option, since higher throughput is observed for bigger block sizes in flash access (experiments on LogData component provided by TinyOS show that flash write speed is 12.99KB/s for a block size of 16B and 42.37KB/s for a block size of 128B). Finally, upon the reception of a `stop record` command, the service updates flash metadata and file metadata and commits them into EEPROM.

At the beginning of a replay stage, when a `start replay` command is received, the service takes several initial steps: (i) it loads into RAM the metadata of the corresponding file, whose run ID matches the one indicated in the command; (ii) to ensure data consistency, before replaying, it further verifies the application ID contained in file metadata against the one indicated by the user or produced by the preprocessor; (iii) it loads log items from flash to fill up the log buffer. After initialization, the service marks the current time as the reference time and starts to replay logged events. During replay, the service automatically loads data from flash to fill up the buffer whenever the buffer is half-empty. The service discards all `status` log items and replays `event` log items one by one. It dequeues the first `event` log item from the buffer,

Table 4.2: Implementation characteristics of EnviroLog components

Component	Language	Code length (lines)	Data memory (bytes)
Preprocessor	Perl	873	
Stage controller	nesC	604	single-hop:46, multi-hop:137
Record and replay	nesC	758	54+buffer size

sets a timer based on the timestamp contained in the item, and upon the expiration of the timer, issues the corresponding function call. Then it proceeds with the next log item. The expiration time $T_{expiration}$ is calculated as follows:

$$T_{expiration} = \frac{T_{reference} + T_{timestamp} - T_{current}}{S_{replay}} \tag{4.3}$$

where $T_{reference}$, $T_{timestamp}$, and $T_{current}$ represent the reference time, the timestamp, and the current time respectively, and S_{replay} represents replay speed. As discussed before, S_{replay} is one of the parameters of `start replay` commands to speed up or slow down replay. Another way to calculate the expiration time is to take the difference between the timestamps of two successive events and divide it by S_{replay}. It is deserted because it makes time accuracy of the latter event always depend on the former one, and consequently accumulates errors over time. The replay stage ends when the entire file is processed or a `stop replay` command is received.

During the retrieve stage, the service simply discards all `event` log items. For `status` log items, it extracts the variable values and sends them to the stage controller, which then displays the data for end users.

4.3 Implementation

In this section, we describe an implementation of EnviroLog, which has been fully tested on Mica2 and XSM hardware platforms. This implementation is expected to work on Mica and Mica2Dot as well. The common features of hardware platforms that this version of EnviroLog operates on are (i) 4KB EEPROM inside the micro-controller and (ii) 512KB external data flash. EnviroLog is implemented on TinyOS 1.x, a popular operating system for the aforementioned hardware platforms. Table 4.2 lists the implementation characteristics of different components in EnviroLog.

4.3.1 Preprocessor Implementation

The preprocessor is essentially a translator that takes a TinyOS application annotated with EnviroLog annotations as input and outputs its corresponding version with the EnviroLog service integrated. Figure 4.5 depicts the main steps of this translation:

- Step 1: For modules annotated by `/*LOG_MODULE*/`, the preprocessor enumerates all function calls issued by these modules and annotates them by `/*LOG_FUNCTION*/`.
- Step 2: The preprocessor scans the entire application to search for all function calls that are annotated by `/*LOG_FUNCTION*/`. A unique log ID is assigned to address each of them. The clause that issues an annotated function call is replaced by a segment of code that (i) issues the function call only when the application is not at the replay or retrieve stage, and (ii) records the call's log ID and parameters during the record stage using APIs of the record and replay service.
- Step 3: The preprocessor also creates event handlers to handle replay requests from the record and replay service. For each replay request, it generates code to extract parameters and execute the corresponding function call.
- Step 4: after scanning the entire application, the preprocessor enumerates EnviroLog annotations in the form of `/*LOG_VARIABLE: variable_name*/`. The preprocessor assigns unique log IDs to them, and translates each of them into a segment of code that records the variable value and its log ID using APIs of the record and replay service.

The preprocessor then automatically wires necessary components (e.g., the record and replay service) into the resulting application to complete the integration of EnviroLog.

4.3.2 Stage Controller Implementation

Figure 4.6 depicts the field deployment of a system to use the stage controller. End users type-in stage control commands through the PC. The server node, which is connected to the PC through a serial cable, forwards the commands to the field. The client nodes in the field propagate the commands throughout the field, and execute them immediately or in a two-phase manner.

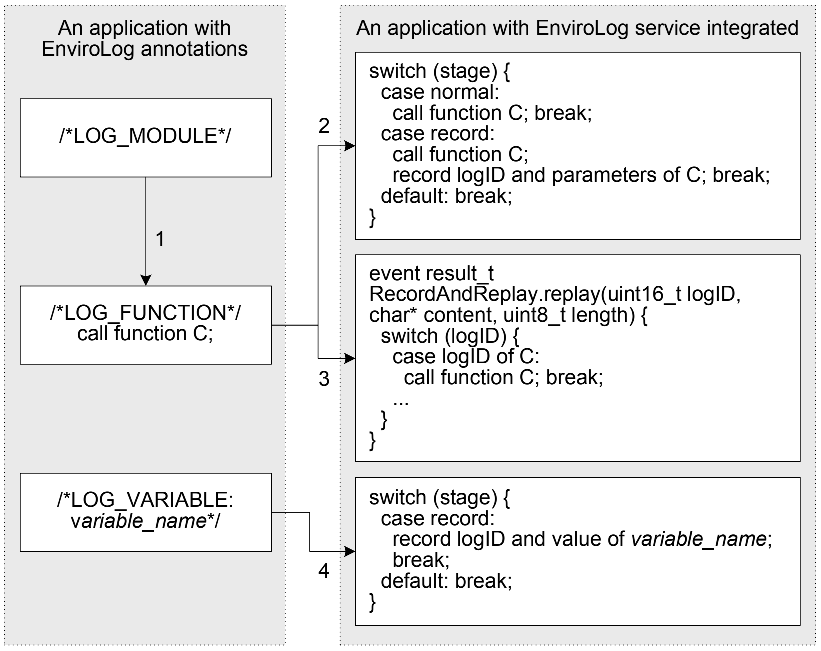

An application with EnviroLog annotations
/*LOG_MODULE*/
1
/*LOG_FUNCTION*/ call function C;
/*LOG_VARIABLE: variable_name*/
2
3
4
An application with EnviroLog service integrated
switch (stage) { case normal: call function C; break; case record: call function C; record logID and parameters of C; break; default: break; }
event result_t RecordAndReplay.replay(uint16_t logID, char* content, uint8_t length) { switch (logID) { case logID of C: call function C; break; ... } }
switch (stage) { case record: record logID and value of variable_name; break; default: break; }

Figure 4.5: Translation steps of the preprocessor

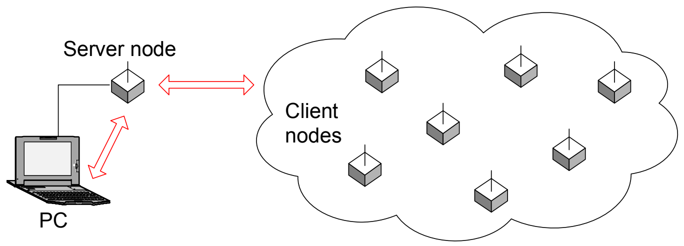


Figure 4.6: A field deployment to use the stage controller

We implement a simple Java tool on the PC to interact with end users. This tool has several functionalities such as: (i) encoding stage control commands into messages, (ii) injecting the messages into the server node through the serial port, (iii) receiving messages from the serial port, and (iv) displaying retrieved variable values to end users during the receive stage. The server node, running the `TOSBase` application provided by TinyOS, forwards messages between the PC and client nodes. The client nodes run the system with EnviroLog integrated, which includes a stage controller component.

The implementation of the single-hop version of the stage controller is simple. Commands are immediately executed upon reception and execution results, if any, are sent back to the server node through one-hop unicast. The multi-hop version that supports two-phase command execution needs more functionality. First, it contains a time synchronization service modified from multi-hop FTSP [68]. Multi-hop FTSP utilizes periodic flooding of synchronization beacons to perform continuous re-synchronization. We modify it to stop the periodic beacons at the end of the first phase; otherwise, these beacons may interfere with the system and change its behavior. Second, to conserve energy, commands are piggybacked onto periodic synchronization beacons. Client nodes remember the command when they receive the first synchronization beacon. The periodic nature of synchronization beacons also makes the dissemination of commands robust to sporadic message losses. Third, the service incorporates a simple routing algorithm to collect execution results from client nodes. The routing service we implement is similar to directed diffusion [44]. Although the primary purpose of synchronization beacons is to synchronize clocks of client nodes, they also serve as interest beacons for client nodes to set up reverse paths to the server node. Later on, execution results can be sent back to the PC along those paths. Note that, the modified Multi-hop FTSP and the simple routing service become parts of the EnviroLog service only if users configure the stage controller as a multi-hop one before the application gets processed by the preprocessor. They are only invoked during the two-phase command execution and, therefore, are independent of any time synchronization or routing service used by user applications.

4.3.3 Record and Replay Implementation

The record and replay service provides a set of APIs to interact with application components and other EnviroLog components. It is implemented in one big component named `RecordAndReplayC`. Figure 4.7 illustrates the `RecordAndReplay` interface provided by the `RecordAndReplayC` component and Figure 4.8 depicts the interactions between this component and other components in the system. Application components (already processed by the preprocessor) call the command `record` to log function calls as well as variable values. During the replay stage, the record and replay component signals the event `replay` to request application components to execute the corresponding function calls. Another event `retrieve` is signaled to transfer retrieved variable values to the stage controller component, which then communicates the data back to the server node and, finally, to the PC. The stage controller component interacts with the record and replay component by issuing the command `executeCommand` to execute stage control commands from end users.

The record and replay component relies on several TinyOS modules: `clock`, `timer`, `EEPROM access`, and `flash access`. The clock component is used for the purpose of timestamp calculation. The timer

```
interface RecordAndReplay {
  command result_t record(uint16_t logID, uint8_t logType, char* content, uint8_t length); ①
  event result_t replay(uint16_t logID, char* content, uint8_t length);                    ②
  event result_t retrieve(uint16_t logID, char* content, uint8_t length);                  ③
  command result_t executeCommand(uint8_t name, uint8_t stage, char* parameters);          ④
}
```

Figure 4.7: Interface of the record and replay component

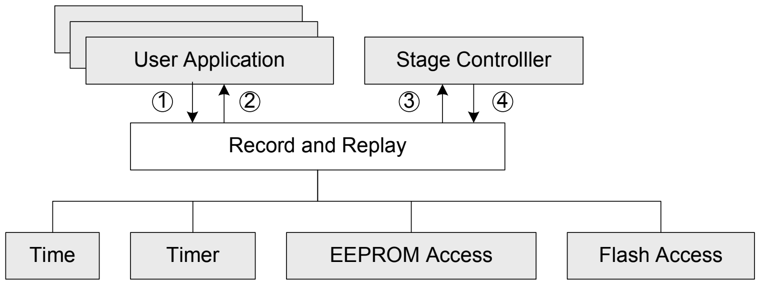


Figure 4.8: Interactions between the record and replay component and other components

component is utilized during replay to issue logged function calls in their original time sequence. Flash access and EEPROM access components are employed to read/write logged data and metadata.

4.4 Performance Evaluation

To evaluate the effectiveness and efficiency of the recording and replay service provided by EnviroLog, we integrate EnviroLog into several sample applications delivered with TinyOS, upload them onto XSM motes, and carry out a series of empirical experiments. The XSM platform is an extended version of Mica2 motes, featuring improved peripheral circuitry, improved antenna and new types of sensors. The purpose of these microbenchmark experiments is to characterize the performance of EnviroLog by illustrating its maximum recording period, throughput, overhead, and replay accuracy. We also carry out more complex machrobenchmark experiments on a practical surveillance system called Vigilnet [37], which is to be presented in Chapter 5.

In this section, we pick several sample applications provided by TinyOS and run a series of microbenchmarks to show how EnviroLog performs in terms of maximum recording period, throughput, overhead, and replay accuracy. These results, collected from simple applications, provide insights into the relevant basic aspects of EnviroLog's expected performance. Larger, more realistic applications are investigated later (Chapter 5), highlighting higher-level performance aspects.

4.4.1 Maximum Recording Period

Due to limited storage space, EnviroLog cannot continuously record an infinite number of events. The term *maximum recording period* describes how long EnviroLog is able to continuously record. Maximum recording period RP_{max} depends on three factors: flash size S_{flash}, expected event interval $E\{eventInterval\}$ and expected log item length $E\{logItemLength\}$. The expected event interval indicates the average length of the time intervals between successive events. The expected log item length indicates the expected flash space a recorded event occupies. The maximum recording period is calculated as follows:

$$RP_{max} = \frac{S_{flash}E\{eventInterval\}}{E\{logItemLength\}} \tag{4.4}$$

Setting the flash size to be 512KB, Figure 4.9 depicts the maximum recording period for different expected event intervals and log item lengths. Typical raw sensing data, if generated at 10Hz, can be recorded for about 90 minutes, which is usually long enough for purposes of debugging or tuning sensor drivers.

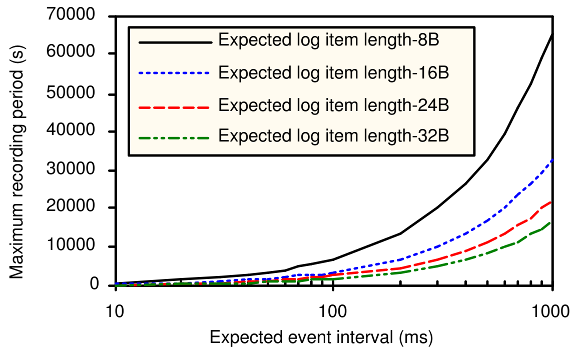


Figure 4.9: Maximum recording period for different expected event interval and log item length

4.4.2 Throughput

The throughput of EnviroLog (i.e., how fast EnviroLog is able to record) is evaluated based on the `Blink` application. `Blink` sets a periodic timer and toggles the red led when the timer fires. EnviroLog is used to record the toggling of the red led. The log item length equals the length of the log item header (7 bytes) since the toggling event contains no parameters. To make the scenario more realistic, the occurrences of toggling events are modeled as a Poisson process by making the time intervals between successive events exponentially distributed with the density function

$$f(x) = \lambda e^{-\lambda x} \tag{4.5}$$

where λ is the expected event interval.

To increase throughput, EnviroLog buffers events in memory before committing them to flash. We repeatedly change the expected event interval and buffer size, compile and download changed `Blink` onto a XSM mote and measure the success ratio of recording operations. Figure 4.10 illustrates the experimental results. Each point in this figure is the average of at least 10 runs to achieve a high confidence level. As shown in Figure 4.10, a higher success ratio is observed for lower event rates and bigger buffers. For events at 10Hz, a 128-byte buffer is enough to ensure a 100% success ratio.

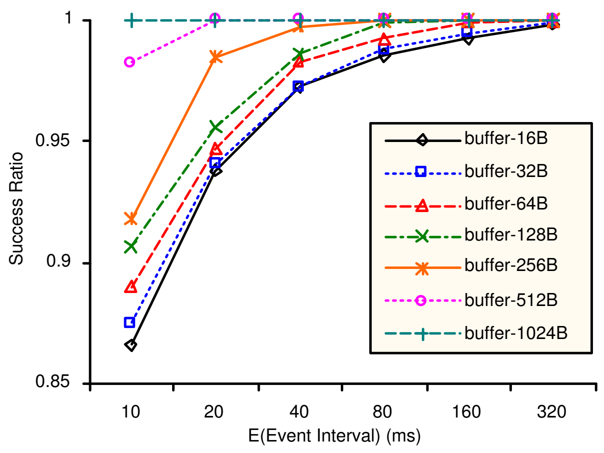


Figure 4.10: Success ratio of recording operations for different expected event rate and buffer size

4.4.3 Overhead

EnviroLog introduces a certain overhead, which may affect the runtime behavior of the original application. It must be verified that EnviroLog does not change the behavior of the original application dramatically during recording; otherwise, the replay of recorded behavior (which is dramatically different from the original behavior) becomes meaningless.

We try to quantify the overhead and its effect through an example application `CntToLedsAndRfm`. The application maintains a counter on a 4Hz timer and sends out the value of the counter by broadcast on each increment. The execution of the command that outputs the value of the counter by messages is recorded and replayed by EnviroLog. We change the message format to contain the `send` time of the current message and the `sendDone` time of the previous message in addition to the value of the counter, so that the time period to send a complete message, defined as *sending delay*, can be calculated by overhearing these messages. The modified `CntToLedsAndRfm` application is run on one XSM mote. Another XSM mote works as the server node of the stage controller to control the stage as well as overhear messages.

This experiment compares the sending delay during a normal stage with the one during a record stage to quantify the overhead of recording operations. Figure 4.11 depicts the cumulative distribution function of sending delay for 1000 messages during the normal stage and 1000 messages during the record stage. A longer sending delay is observed during the record stage compared with the normal stage, which indicates the overhead of recording operations. However, the effect of the overhead is trivial and acceptable. It is observed that the 95% confidence interval of the sending delay during the recording stage drifts only 0.4ms from the one during the normal stage.

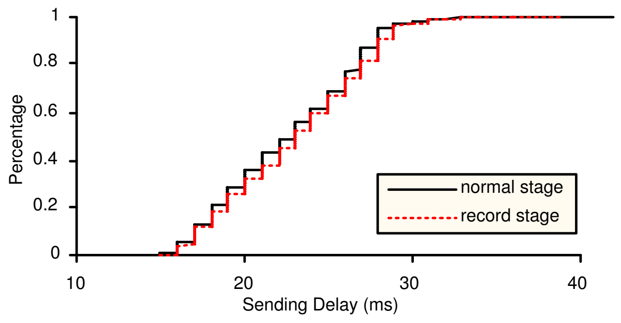


Figure 4.11: Comparison of sending delay between normal stage and record stage

4.4.4 Replay Accuracy

In this experiment, we use the same modified version of `CntToLedsAndRfm` as in the previous experiment. We first log about 100 commands by EnviroLog and overhear the messages to remember their send time. The remembered send time is actually the time when each logged command is executed. Then, we replay those commands 20 times. We calculate the difference between the average send time during the replay stage and the original send time during the record stage and depict its cumulative distribution in Figure 4.12. The results reflect how accurate the replay service is. As Figure 4.12 shows, the average error is less than 1ms.

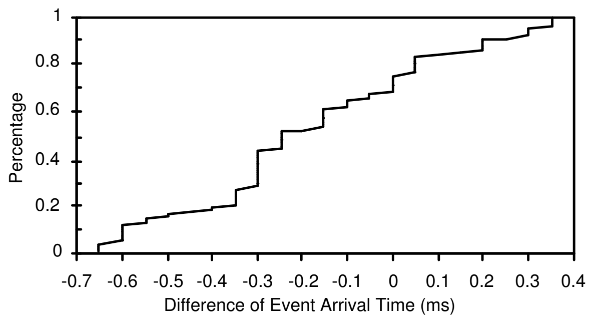


Figure 4.12: Difference of send time between record stage and replay stage

4.4.5 Concluding Remarks

Results from the series of microbenchmarks above validate the effectiveness of EnviroLog. EnviroLog is able to record and replay high frequency events if assigned a big enough buffer (e.g., recording 10Hz events with a 128-byte buffer in the `Blink` application). Its recording operations bring little overhead (e.g., adding only 0.4ms delay in the `CntToLedsAndRfm` application). It also replays events accurately (e.g., the average difference between timestamps of recorded events and replayed events is less than 1ms).

4.5 Conclusions

With the increasing popularity of wireless sensor networks, an increasing number of realistic applications employing large systems of sensor devices emerge. Although the initial development and debugging of these applications can be aided by simulators, in-field tests still have to be conducted at a later stage due to typical discrepancies between simulation results and empirical measurements. In this chapter, we present the design, implementation and evaluation of EnviroLog, an asynchronous event record and replay service that improves repeatability of environmental events for in-field testing of distributed event-driven applications. The friendly user interface of EnviroLog allows users to integrate and utilize the service merely by inserting annotations into their applications and learning a few operation commands. Based on several sample applications of TinyOS, we validate the effectiveness of event recording and replay. In a later chapter (Chapter 5), we will demonstrate the usefulness of EnviroLog in various aspects of in-field tests such as performance tuning without physically generating events, runtime status collection without extra hardware, and virtual velocity simulation.

The potential uses of such service are not limited to what we have discussed. EnviroLog can be further extended to perform remote replay (recording events in environment A while replaying them remotely in environment B), and off-site replay (recording events on sensor devices while replaying them in simulators), which are on our agenda for future work on EnviroLog.

Chapter 5

Integrated Systems

5.1 Introduction

To demonstrate the efficacy of EnviroSuite, we apply it to multiple application scenarios, including both target tracking and environmental monitoring, to evaluate different subsystems of the framework. This chapter presents empirical results collected during the in-filed deployments of these applications.

In previous chapters, we carry out experiments on either simulators or small table-top testbeds. Such experiments are more repeatable and controllable to achieve fine measurements of the performance. However, the inconsistency between simulators (isotropic radio range, isotropic sensing range, and faked environmental events), indoor testbeds (faked radio range, and faked environmental events), and outdoor deployments (non-isotropic radio range, non-isotropic sensing range, and real environmental events) demands us to move to in-field experiments under physical environments. Therefore, this chapter supplies a more realistic and extensive exploration of the performance of EnviroSuite.

However, we did not include an experimental comparison between EnviroSuite and other high-level sensor network programming frameworks. This is, in part, due to the difficulty in porting application code across the different systems. If applications are re-implemented (as opposed to ported), it is hard to separate the effects of application-level implementation decisions from the inherent strengths and weaknesses of the underlying programming frameworks when interpreting performance comparison results.

The rest of the chapter presents the two applications built upon EnviroSuite and empirical results collected during the in-field deployments. Section 5.2 presents Vigilnet as an example of target tracking applications, and Section 5.3 presents EnviroMic as an example of environmental monitoring applications.

5.2 Vigilnet: Energy-efficient Surveillance

We applied EnviroSuite to build an energy-efficient surveillance system, called Vigilnet [37][36], subsequently transitioned to the Defense Intelligence Agency (DIA). The primary goal of the system is to detect, classify and track one or multiple moving targets, which can be either cars, persons or persons carrying a ferrous object (suggestive of a weapon). We utilize Vigilnet for two purposes:

First, Vigilnet serves as an evaluation platform for the core library of EnviroSuite: object management algorithms. Vigilnet utilizes event objects to track various targets. The event objects communicate the estimated target locations to function objects hosted by base stations, where complex time and temporal correlation algorithms are run to filter out false alarms. Therefore, the tracking performance of Vigilnet can indirectly reflect the effectiveness of EnviroSuite to maintain a unique mapping between physical targets and logical objects.

Second, we use Vigilnet as a demonstration platform to showcase the potential uses of EnviroLog in practice. Results collected from in-field experiments of Vigilnet demonstrate how EnviroLog effectively and accurately replays events on demand. By enabling low-cost performance tuning and evaluation, runtime status collection, and virtual velocity simulation, EnviroLog greatly simplifies the in-field testing of Vigilnet.

5.2.1 Overview of Vigilnet

Vigilnet is implemented on top of TinyOS. Figure 5.1 shows the layered architecture of Vigilnet. Components colored in dark grey are middleware services adapted from EnviroSuite.

Time synchronization (Time Sync), localization (Localization), and communication (MAC, Robust Diffusion Tree, Asymmetric Detection, and Report Engine) services constitute the lower-level components that are the basis for implementing higher-level services. Power management (Radio-Base Wakeup, Sentry Service, Power Mgmt, and Tripwire Mgmt), target classification (Sensor Drivers, Frequency-filter, Continuous Calibrator, Classification, and False Alarm Filtering Engine) and tracking (Group Mgmt, Tracking, and Velocity Regression) comprise main higher-level services. Target classification detects and classifies three types of targets with the help of collaborative group management provided by EnviroSuite. Tracking components are responsible for estimating target positions and calculating target velocities.

Overall the system consists of 21,457 lines of source code, among which 2,884 are contributed by EnviroSuite. The executable binary of Vigilnet occupies 85,926 bytes of code memory and 3,154 bytes of

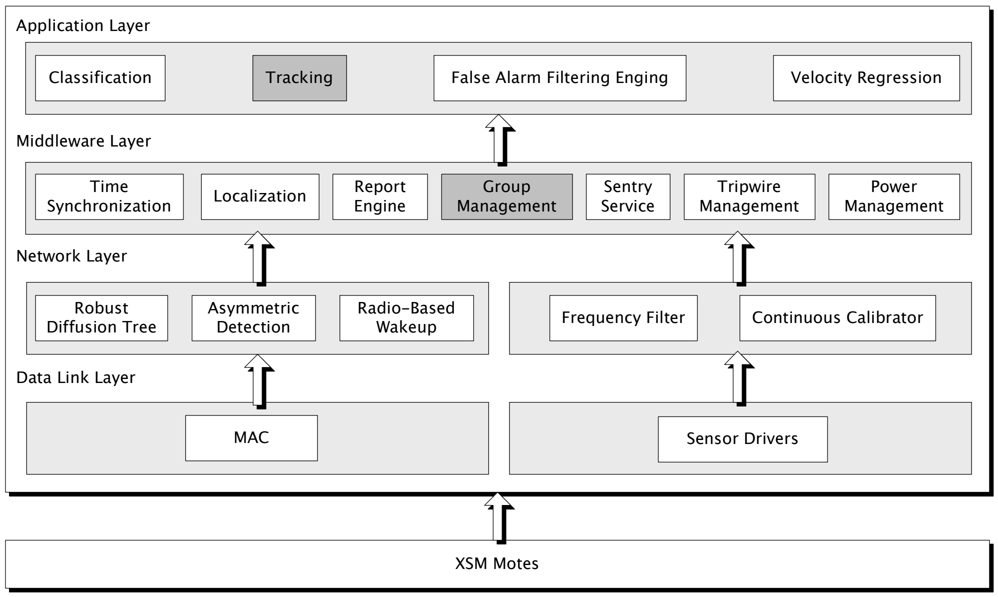


Figure 5.1: System architecture of Vigilnet

data memory, which can easily fit into XSMs equipped with 4KB data memory and 128KB code memory. Vigilnet has been transited to the DIA for use in special operations (SOCOM).

The hardware used to test Vigilnet are XSM motes, which extends the Mica2 platforms by improved peripheral circuitry, new types of sensors and better enclosures. Three types of sensors are utilized in Vigilnet: magnetic sensors to detect moving ferrous objects, acoustic sensors to detect noise of vehicles, and passive infra-red motion sensors to detect any moving objects including persons.

5.2.2 Tracking Performance Evaluation

As Vigilnet utilized the object management algorithms of EnviroSuite to keep track of targets, the efficiency of such algorithms greatly affects the tracking performance of Vigilnet. In this section, we present empirical results on tracking performance of Vigilnet as an indirect measurement of performance of object management algorithms.

Methodology

In December 2004, in the process of technology transition to DIA, we deployed 200 XSM motes running the Vigilnet system on sandy and grassy roads with a 3-way intersection and collected performance data in field tests. The primary goal of the field test is to evaluate system ability to detect, classify and track one or

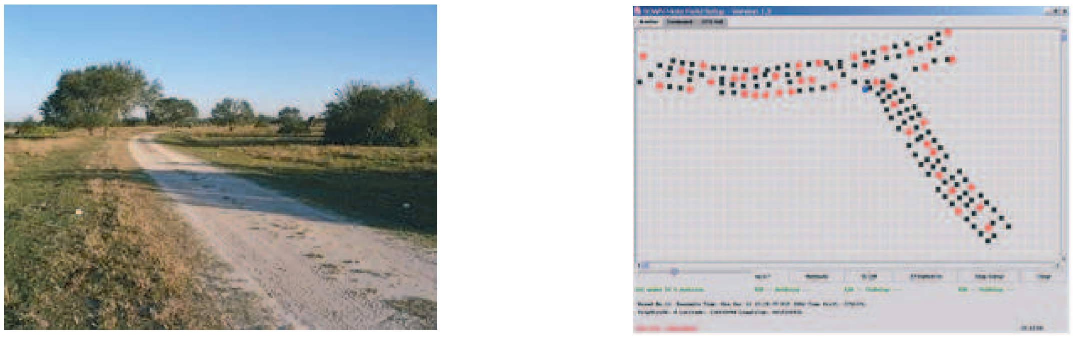


Figure 5.2: System deployment of Vigilnet

Table 5.1: Tracking performance for varied target velocities

Target velocity	Tracking performance
5 mph	successful
10 mph	successful
20 mph	successful
30 mph	successful
35 mph	partially successful

multiple moving targets, which can be either SUVs, persons or persons carrying a ferrous object (suggestive of a weapon). Figure 5.2 depicts the deployment of the system. Nodes are approximately deployed in a grid 10 meters apart, covering one 300-meter road and one 200-meter road. Each rectangular dot represents one XSM mote in the field. Several base stations were deployed. Some nodes are missing in the GUI because they are turned off to emulate failures.

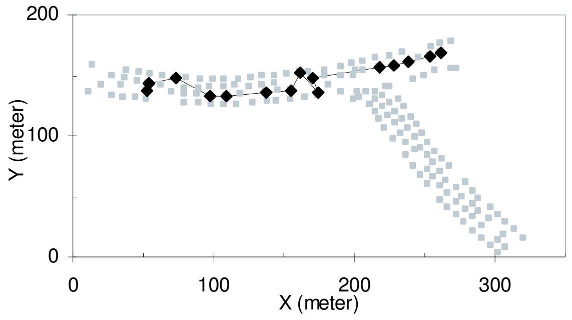
Evaluation Results

Table 5.1 shows the tracking performance of Vigilnet. As is seen, the Vigilnet system tracks targets up to 35mph. Note that, the success of tracking is an end-user metric measured by the accuracy of position and velocity calculations, which depends on several factors besides EnviroSuite object management protocols. A track is said to be successful only when the final calculated velocity is within a 20% error. Due to the limited length of the field and the fixed report rate (once every 3 seconds), velocity calculation does not perform well when the velocity reaches 35mph. However, the tracking performance of EnviroSuite itself is actually better than the reported results for the integrated system.

Table 5.2 shows in more details the tracking performance of the Vigilnet system. As is seen, tracking errors are between 5.5 meters and 7.5 meters. These results were collected with 20% of the nodes randomly

Table 5.2: Tracking performance of Vigilnet with EnviroSuite

Target type		Avg. tracking error	Std. dev. of tracking errors	Actual vel.	Calculated vel.
walking person		6.19 meter	3.28 meter	3±1 mph	2.9 mph
running person		6.67 meter	3.89 meter	7±1 mph	6.9 mph
vehicle		7.06 meter	3.98 meter	10±1 mph	10.5 mph
vehicle		5.91 meter	3.02 meter	20±1 mph	23.5 mph
two vehicles	1	5.58 meter	4.76 meter	10±1 mph	9.2 mph
	2	6.33 meter	3.52 meter	10±1 mph	9.9 mph



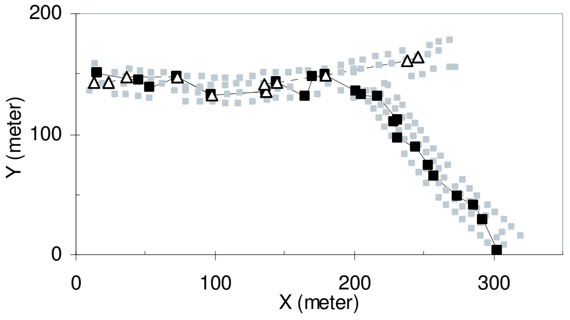


Figure 5.3: Tracking trajectories for one vehicle and two vehicles

turned off to emulate failures. In all listed targets whose velocities vary from 3mph to 20mph, the maximum error of velocity calculation is less than 10%, which reflects the good tracking performance supplied by EnviroSuite.

To give a more concrete view of the tracking performance of EnviroSuite, Figure 5.3 shows the tracking trajectories for the following scenarios: (i) one vehicle drives across the field from left to right; (ii) two vehicles keep a distance of about 50 meters before they separate (the first one goes from left to right and turns right at the intersection and the second one goes from left to right). In the one-vehicle-tracking case, the rugged trajectory in the center of the horizontal road shows explicitly that existing node failures do affect tracking accuracy. The two-vehicle-tracking case proves the ability of EnviroSuite to track multiple targets with the same sensory signatures as long as they keep a distance (50 meters) that is more than half an object resolution (set to 30 meters in the system). This was deemed sufficient by the client for operational use.

Concluding Remarks

Collected results show that the maximum trackable speed of EnviroSuite under this specific deployment is near 35 mph. Given our 10-meter-apart grid deployment and 20% node failures, tracking errors are still as small as about 6 meters and the maximum error in velocity calculation doesn't exceed 10%. These results from physically deployed systems validate that EnviroSuite is practical, effective, and efficient on current hardware with limited communication and sensing capabilities.

5.2.3 Evaluation of EnviroLog

We show the various functionalities of EnviroLog by using the recording and replay service to tune and evaluate performance of Vigilnet, to collect its runtime status and to replay targets with virtually increased or decreased velocities.

In this section, we first introduce the experimental methodology, including hardware, software and deployment scenarios. Then, we show the effectiveness of EnviroLog by recording and replaying different types of targets. Finally, we show how EnviroLog aids in-field tests of Vigilnet in various ways, including performance tuning and evaluation, runtime status collection, and virtual velocity simulation.

Methodology

As stated before, Vigilnet is targeted to detect, classify and track various events of interest in real-time through in-network processing. It takes environmental targets as inputs, applies multiple levels of processing before outputting results to end users. The lowest layer is sensor drivers which sample raw data from sensors and, if any target of interest is detected, signal detection results to higher layers. The layer above sensor drivers is object management algorithms from EnviroSuite that dynamically organize nodes in the vicinity of targets into local objects, and send the aggregate states (such as target position estimates) of these objects to function objects located at base stations. The highest layer is located in base nodes, where aggregate reports are further processed to extract properties such as target velocities. The highest layer outputs target types (vehicles or persons), trajectories and velocities to a GUI which displays results to end users.

An obvious use of EnviroLog in this system would be to debug sensor drivers, to tune their sensor data processing parameters (e.g., various filters), or to compare their different versions by recording and replaying raw sensing data on individual nodes. However, what is more interesting is to see how EnviroLog aids the in-field tests of higher layers that involve coordination among multiple nodes in a multi-hop wireless network. Towards that end, we insert EnviroLog between the sensor driver layer and the group management layer, record outputs of sensor drivers rather than raw sensing data, and focus on the behavior of layers higher than the sensor drivers.

Vigilnet employs three types of sensors: magnetic, acoustic and motion sensors. Their drivers interact with higher layers by signaling instances of the following event:

```
event result_t detected(TargetConfidence confidences);
```

where `TargetConfidence` is an array of integers, each representing the possibility of a certain target type.

To integrate EnviroLog into the system, we simply insert `/*LOG_FUNCTION*/` before each clause that signals the event, and process the code using the preprocessor before compiling the system. To collect empirical data, we download the system onto 37 XSMs. We deploy the XSMs approximately 5 meters apart on both sides of a driveway, as shown in Figure 5.4. The triangle marks the position of the base node connected to a laptop.

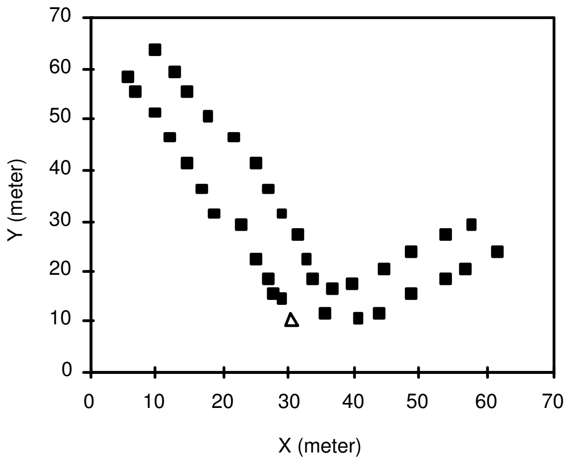


Figure 5.4: System deployment of Vigilnet and EnviroLog

Effectiveness

To evaluate the effectiveness of recording and replay, we first set the system to be at record stage, and physically generate targets by jogging or driving through the driveway. The trajectories and calculated velocities for the jogging person and the vehicle are shown separately in Figure 5.5(a) and 5.5(d). Later on, we switch the system to its replay stage to virtually replay the jogging person and the vehicle. Figure 5.5(b) and 5.5(c) shows two different replays of the person, while Figure 5.5(e) and 5.5(f) shows two replays of the vehicle.

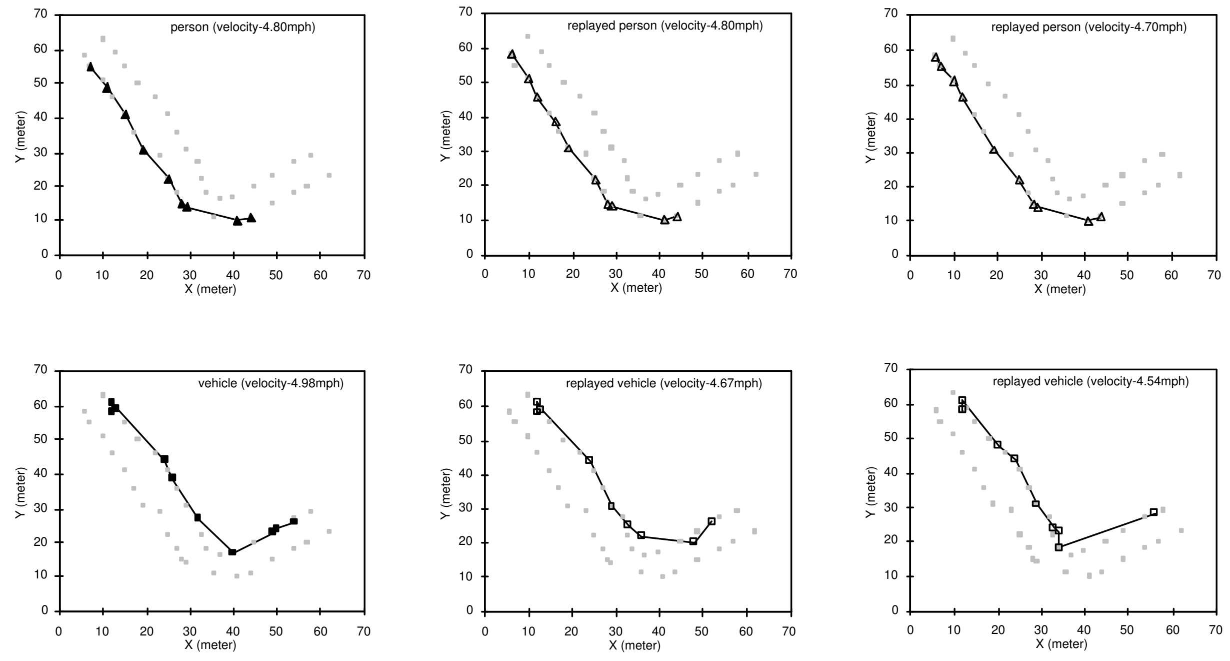


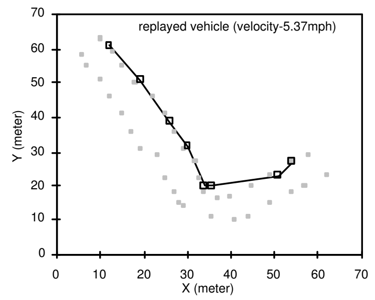
Figure 5.5: Trajectories and calculated velocities for physical and replayed targets

As is seen for both the person and the vehicle, the trajectories of real targets and replayed ones are very close, and the differences of their calculated velocities are below 0.5mph. This observation verifies the effectiveness of EnviroLog. However, outputs are not exactly the same, which is expected considering the variability in layers above sensor drivers. For example, communication delays may change due to randomness in the MAC layer especially when multiple neighboring nodes request to send simultaneously. This also explains why replay of the person is more accurate than that of the vehicle. A person is usually detected by only one node while vehicles can be detected by multiple nodes simultaneously, which causes those nodes to send detection reports to leaders simultaneously. The possibility that these detection reports are always received by the leader in the same order during different runs is very low, which leads to the minor differences among observed target trajectories.

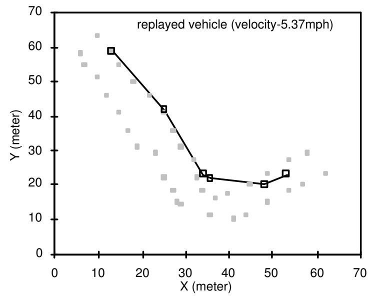
Potential Uses - Performance Tuning and Evaluation

A big portion of wireless sensor network applications are outdoor applications that detect targets or monitor environments. It is difficult to evaluate such applications by simulators, which either can't simulate environmental inputs or can't realistically simulate them. The environments are much more complex than what simulators can model. Even the modeling of the magnetic field near a vehicle is extremely challenging due to the uneven and unknown distribution of metals inside the vehicle. The lack of realistic sensing models makes in-field testing a necessary step before the real deployment of most applications involving target detection or environmental monitoring. EnviroLog makes in-field tuning and evaluation much easier as shown by the following experiments.

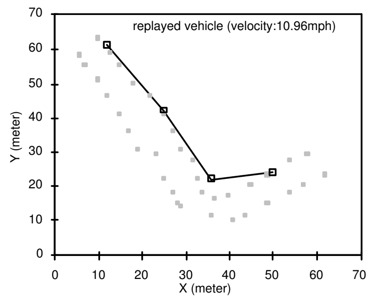
In Vigilnet, the group management layer used to group nodes that detect the same target has a tunable parameter called *DOA* (*degree of aggregation*). It is used to eliminate sporadic false positives in target detection. Group leaders do not report the detection of a target to the base station until the number of nearby nodes that detect the target reaches DOA. Higher DOA filters out more false positives, thus reducing the number of reports from leaders. However, too high DOA results in false negatives. To find out the proper DOA, we have to drive the vehicle or walk through the field multiple times while tuning this parameter. EnviroLog provides an alternative way to tune DOA with less overhead. We drive a vehicle or walk once to record the environmental inputs, and then replay them multiple times with different DOA values. Figure 5.6 shows target trajectories for different DOA settings. As is seen, a higher DOA results in fewer trajectory points, thus more inaccurate calculated velocities. When DOA reaches 4, the calculated velocity (10.96mph) is far away from the ground truth (5$\pm$1mph).



(a) DOA = 2



(b) DOA = 3



(c) DOA = 4

Figure 5.6: Trajectories and calculated velocities for different DOA

We also use EnviroLog to log and retrieve the number of aggregate reports during runtime. Figure

5.7 depicts the cumulative distribution of the number of aggregate reports for different DOA values. As expected, a higher DOA leads to fewer aggregate reports. These results suggest that DOA values of 1, 2 and 3 are acceptable settings, though a higher value that leads to less communication overhead is more preferred.

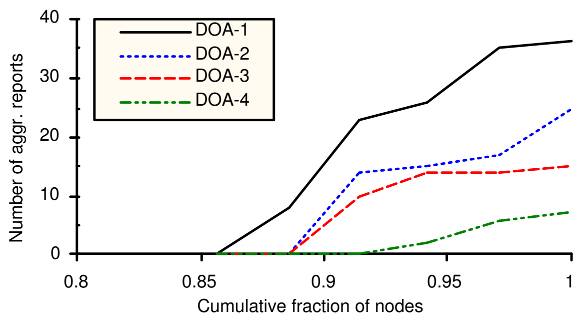


Figure 5.7: Cumulative distribution of number of aggregate reports for different DOA

Potential Uses - Runtime Status Collection

During development and testing, batteries are often depleted due to frequent experiments. One drawback of XSM motes is that batteries can not be measured or replaced without opening the package by unscrewing 4 screws. Vigilnet usually operates at the scale of hundreds of XSMs, which makes it extremely painful to check whether each node has a high-enough voltage. Runtime status recording and retrieve supported by EnviroLog provides a simple solution for this problem. Voltage values of nodes can be logged and retrieved after each in-field test to find out those with low voltage, whose batteries then can be replaced before next test. Figure 5.8 shows the cumulative distribution of voltage values before and after the whole set of macrobenchmarks, which are collected through EnviroLog.

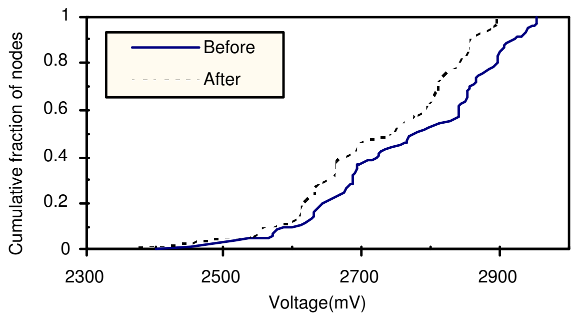


Figure 5.8: Cumulative distribution of voltage values before and after experiments

Potential Uses - Virtual Velocity Simulation

EnviroLog allows users to speed up or slow down the replay of events by setting a replay speed greater or less than 1. Note that changing replay speed is not always meaningful. For example, if sensor drivers pull data at a fixed rate and raw sensing data is logged, replaying at a different speed actually violates the logic of sensor drivers. In Vigilnet, EnviroLog records and replays the outputs of sensor drivers, which are detection events signaled by sensor drivers to higher layers. Replaying them at a higher speed can virtually simulate environmental targets with higher velocities. This experiment replays recorded events using different replay speeds to validate the effectiveness of virtual velocity simulation. Figure 5.9 depicts the calculated velocities for different replay speeds and targets. When the replay speed doesn't exceed 4$\times$, calculated velocities for both the person and the vehicle are close to the ground truth. Higher replay speeds lead to intolerable errors in velocity calculation.

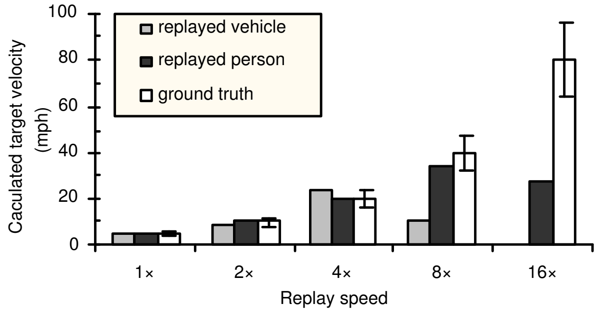


Figure 5.9: Calculated velocities for different replay speed

Concluding Remarks

Results above validate the effectiveness of EnviroLog and demonstrate its great potentials in facilitating in-field experiments. EnviroLog has various uses for in-field tests of large-scale systems, including performance tuning and evaluation, and runtime status collection. EnviroLog can replay events at different replay speed, which can be used to virtually simulate targets moving at different velocities.

5.3 EnviroMic: Distributed Acoustic Monitoring

EnviroMic presents the first implementation of an audio sensor network for recording, storing, and retrieving environmental acoustic traces geared for a prolonged interval of disconnected operation. The purpose is to monitor acoustic events, record sound clips, and collect them for future analysis. It can be used, for example, by avian ecologists to record traces of bird vocalizations in forests to learn about their mating behavior.

Networked deployment of large numbers of low-cost, low-range sensors adds new challenges and opportunities in acoustic recording service design. The main challenges in the design and implementation of EnviroMic are enumerated below. First, the omni-directional nature of acoustic sensing introduces data redundancy when multiple nodes collectively sense the same acoustic source. Therefore, it is imperative to ensure that EnviroMic limits redundant recording such that storage is used more efficiently. Second, since current sensor platforms are severely constrained in CPU bandwidth, they are unable to perform other activities (such as radio communication) concurrently with high frequency sampling. Therefore, it is important to coordinate the recording and other tasks such that they do not interfere. Finally, the high data volume generated by an acoustic source, coupled with the potentially uneven spatial distributions of such sources, may cause some nodes to overflow while others still have available storage space. Hence, EnviroMic must balance recorded data across nodes to eliminate storage hot-spots and to make use of storage capacity that is not in direct vicinity of the frequent sound sources.

To address these challenges, we design and implement EnviroMic based upon the EnviroSuite framework. We utilize event objects to track environmental targets that let out noise (e.g., cars or birds). Such noise is recorded and stored in the network using EnviroStore, and retrieved at the end of deployment for later analysis. A prototype of EnviroMic is built upon MicaZ motes equipped with MTS300 sensor boards [16]. Based on the prototype implementation, we evaluate EnviroMic using both an indoor testbed and an outdoor deployment in a forest. Results collected from these experiments demonstrate the efficacy of EnviroSuite in conserving storage resources, preserving the continuity of recordings, and balancing network load.

5.3.1 Overview of EnviroMic

The target application of EnviroMic is long-term acoustic monitoring, which involves high-frequency sampling and high-volume data storage. The primary concern of the system is to maximize the effective storage capacity of a sensor network, which is maximizing the amount of data a scientist can collect about the environment in a single experiment. It is assumed that the network generally remains disconnected from the outside world during the experiment. Hence, there is need for improving storage capacity. With that design goal in mind, EnviroMic employs mechanisms to reduce storage redundancy and improve balancing of data storage in the network, when energy permits. Below, we briefly introduce the system architecture of EnviroMic and highlight the main EnviroMic subsystems as depicted in Figure 5.10.

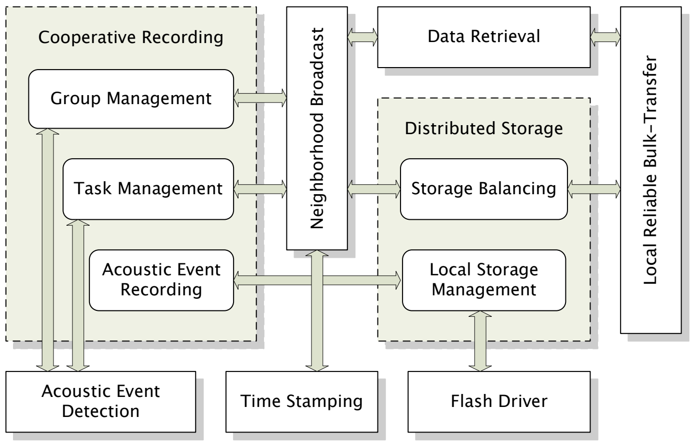


Figure 5.10: System Architecture of EnviroMic

To reduce storage redundancy, EnviroMic employs object management algorithms from EnviroSuite. Event objects are instantiated whenever new acoustic events are detected. We further implement a task management service to rotate the task of recording such events among nodes near the perceived source (for load balancing purposes). We call this whole subsystem the *cooperative recording* subsystem. The cooperative recording subsystem offers two main advantages over uncoordinated local sampling (in which each sensor independently records local acoustic data). First, redundancy is reduced allowing more data to be collected. Second, recording that is perceived to belong to the same continuous acoustic event is coalesced into the same file.

The recorded data is stored via EnviroStore using the log-sequence file interface, attempting to create a single file for each continuous acoustic event. The file is distributed and consists of different chunks residing on different sensors that recorded parts of the event. To improve storage balancing in the network, EnviroStore automatically transfer data from heavy-loaded areas to light-loaded areas when energy permits. EnviroMic also leverages the data retrieval subsystem of EnviroStore to extract data from the network. However, it is not an objective of EnviroMic to guarantee unique file-to-event mapping. It merely does its best in event-file association to reduce redundancy and facilitate indexing of sound clips. This provides the basis for more sophisticated algorithms (executed on powerful base stations after collecting all the data) to extract higher-level information.

We implemented EnviroMic on MicaZ motes running TinyOS. We chose MicaZs since they are good representatives of a large class of off-the-shelf hardware platforms with various sensing and storage capabilities. Though MicaZs are severely constrained in storage capacity and may not be the most appropriate for our specific application, the challenges we experienced in building EnviroMic are not specific to MicaZs, but are in fact general to the large class. The implementation consists of 12 nesC modules, and 10,282 lines of nesC code. The system occupies 61.5KB of code memory and 2.8KB of data memory on MicaZ.

We evaluate EnviroMic using both an indoor testbed and an outdoor deployment in a forest. The acoustic sampling frequency is set to be 2.730kHz throughout the experiments. The indoor testbed consists of 48 MicaZ motes placed as a 8$\times$6 grid with unit grid length 2ft. We use this testbed together with controlled acoustic events described below to achieve repeatability in our experiments so we could perform valid comparisons and empirically determine the effects of certain system parameters. To further understand the performance issues of EnviroMic in realistic environments, we conducted experiments using 36 MicaZ motes in a nearby forest.

5.3.2 In-door Experiments

Efficiency of the cooperative recording subsystem comprises two related properties. First, we want the acoustic event recording to be complete (i.e., no recording gaps). Second, we want to reduce recording redundancy. Our design and implementation of seamless task assignment ensures that recorded data redundancy is almost eliminated.

Figure 5.11 plots the recording periods of the nodes, as well as the acoustic event duration of one instance of our experiments in which task period is 1.0s. Note that not all nodes performed a recording task due to the cooperative task assignment. Recordings happened at different nodes seamlessly. Also note the recording miss at the very beginning when the acoustic target enters the network and nodes are still in the leader election phase.

To better appreciate the efficacy of our cooperative recording subsystem, in Figure 5.12 we present an experiment of recording human voice. In this experiment, a person read out a sentence while moving across

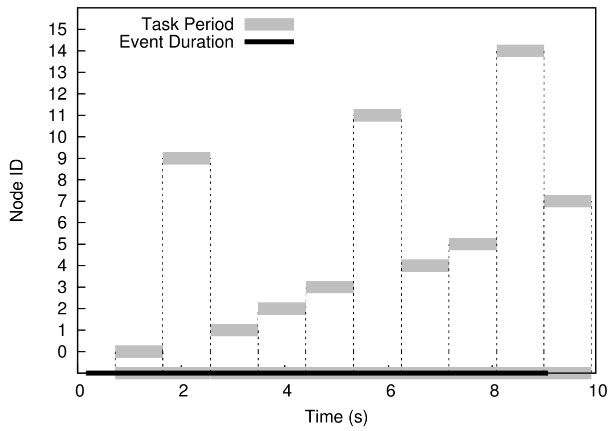
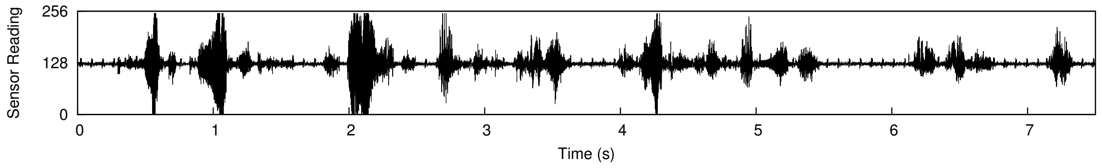
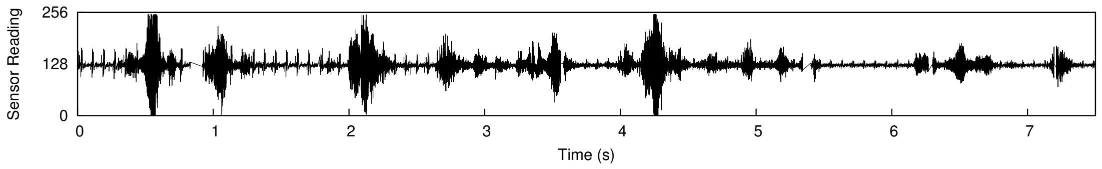


Figure 5.11: One instance of recording a mobile acoustic object



(a) Recorded by a single mote



(b) Recorded by EnviroMic

Figure 5.12: Recording voice of a moving human being

the 7×4 grid of motes at a constant speed of one grid length per second. An extra mote was held by the person during the experiment to record a reference "ground truth". Figure 5.12(a) plots the sensor readings of the mote held by the person, while Figure 5.12(b) plots the sensor readings of all the EnviroMic nodes that performed recording tasks, stitched together based on their timestamps. The visual similarity of the two figures is obvious.

5.3.3 Preliminary Outdoor Deployment

To understand the efficacy and limitations of our current design and implementation of EnviroMic in more realistic environments, we deployed a EnviroMic system that consists of 36 MicaZ motes in a forest (Figure 5.13). On the west side of the forest is a road where vehicles pass by during the day. The experiment

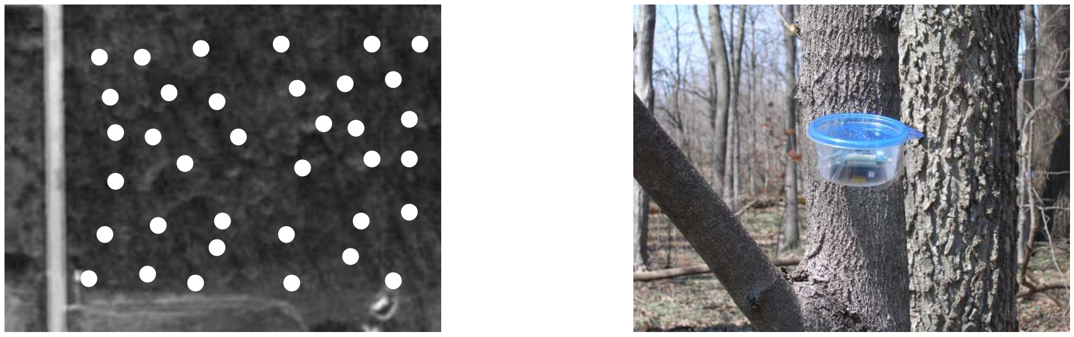

(a) A bird-eye view of the deployment (b) A mote installed on a tree

Figure 5.13: Deployment in a natural forest

was conducted on a day in April 2006. The motes, enclosed in plastic containers, are attached to the trunks of the trees. The deployment area is approximately 105ft×105ft. We were not able to deploy the motes as a grid (to facilitate post-processing of recorded data) because the trees in the forest are in irregular positions. We therefore had to reconstruct the map (Figure 5.13(a)) manually. After the motes were installed, a person holding a mobile device walked around the network to activate the EnviroMic application in each mote. This is to avoid acoustic disturbance caused by installing motes in the containers and attaching them to the trees. We collected data recorded by the motes during a period of 3 hours.

The first set of data of interest is how acoustic events are temporally distributed. Figure 5.14 plots the amount of acoustic data collected by all the sensors at different times. The numbers on the y-axis represent the total amount of recording (in seconds) done by all nodes within one-minute intervals. They are plotted versus time. There are two spikes in the figure. The first spike (11:30-11:40), as we found out later, was caused by people from another department in our university doing an experiment in the forest. The second spike (12:15-12:45) contains some very long events (up to 73 seconds) that, we conjecture, were caused by the motion of heavy agrarian equipment on a neighboring road.

Next, we look at the geographic distribution of the events. From the data stored in the motes, we reconstructed the information on how much acoustic data was generated from which sensors throughout the experimentation period. Mapping node IDs to geographic locations, we plotted a contour graph of total volume of acoustic events (in seconds) shown in Figure 5.15. We can see that there are two high data volume regions. The one on the left side is caused by vehicles passing on the road to the west of the forest. The other region that is rich in acoustic events in the figure roughly matches a trail in the forest.

Since we can not repeat the events, we are unable to compare the performance of EnviroMic with

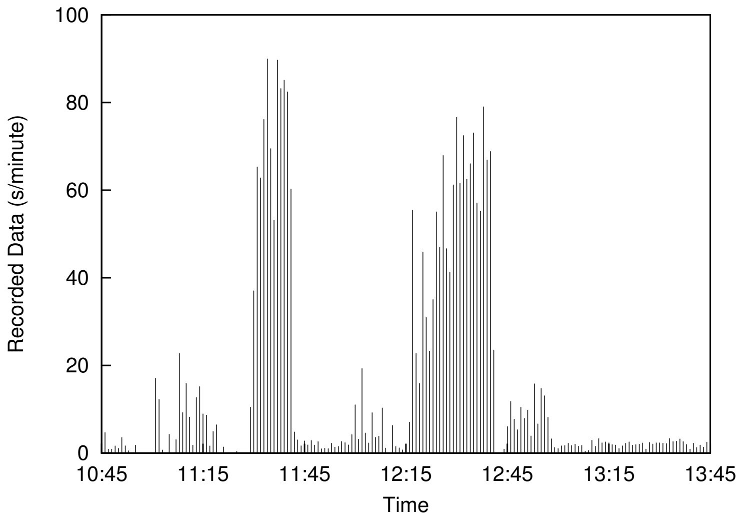


Figure 5.14: Amount of acoustic event data over time

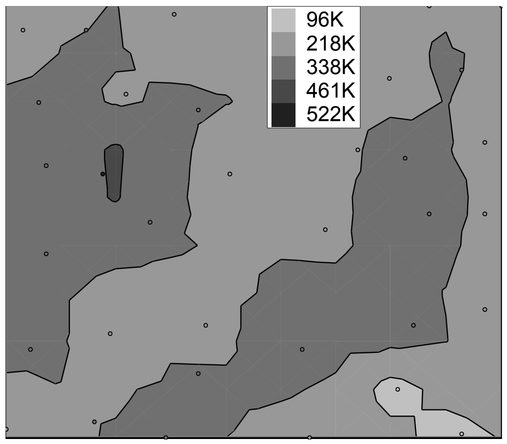


Figure 5.15: Amount of acoustic data, in bytes, generated in different locations

baselines as we did in the indoor experiments. However, it is still interesting to see how data recorded in those acoustically-rich regions migrated to other nodes. We therefore picked the node that recorded the highest volume of data, and plotted the amount of data migrated from that node to other nodes in the network in Figure 5.16. The node that recorded the most data is at coordinate (4, 2) in the rough grid of our deployment. As can be seen in the figure, the node migrated a lot of data to its immediate neighbors, which further migrated some of those data to their neighbors, and so on. This demonstrates the ability of EnviroMic to gracefully handle acoustic event hot-spots.

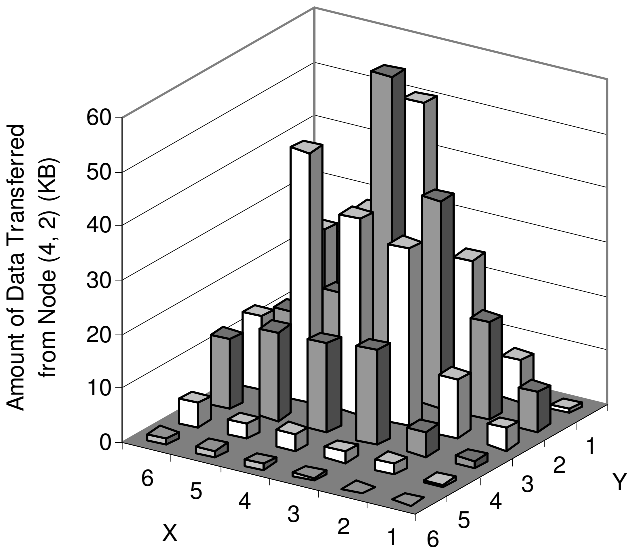


Figure 5.16: Distribution of data migrated to nodes for load-balancing

5.4 Conclusions

The ultimate purpose of the EnviroSuite framework is to provide usable development support to the sensor network community. Therefore, practicability of the framework has to be thoroughly validated. In this chapter, we extensively test and evaluate the work through two practical, integrated systems: Vigilnet and EnviroMic.

First, we presented our successful experience in building a large surveillance system, Vigilnet, upon the object management services provided by EnviroSuite. Experimental results from outdoor deployments of Vigilnet validated the efficacy of event object management in realistic environments. We also experimented with how to utilize EnviroLog to aid in-field testing of Vigilnet. The collected results fully demonstrated the effectiveness of this recording and replay tool as well as its great potentials to enable and simplify a series of testing routines, including performance tuning and evaluation, runtime status collection, and virtual velocity simulation.

We also presented EnviroMic, a novel distributed acoustic monitoring, storage, and trace retrieval system. EnviroMic employs a cooperative recording scheme (adapted from the object management algorithms) and a distributed balanced storage mechanism (i.e., EnviroStore) to address unique challenges arising from high-frequency acoustic sampling and high-volume sensory data storage. Evaluation results drawn from both indoor and outdoor deployments demonstrate the efficacy of our design.

Chapter 6

Related Work

6.1 Programing Paradigms

EnviroSuite opens a new category of distributed programming paradigms. EIP differentiates itself from traditional paradigms such as CORBA [89], Microsoft's COM [71], and remote procedure calls [8] by combining within its programming abstractions objects and events in the physical world.

Several communication and programming models have been proposed for sensor networks in recent years. These include node-based languages, virtual machines, database-centric abstractions, event-based models, and group-based primitives. EnviroSuite is different in that its abstractions are not centered about computational constructs such as queries or sensor groups. Instead, these abstractions are centered around elements of the physical environment.

Node-based languages such as nesC [27] and galsC [12][13] are too low-level since they typically take the sensor node as basic computation, communication and actuation unit. To address this issue, higher-level languages that export logical nodes [33] were proposed to abstract away from physical sensors. EnviroSuite successfully raises the abstraction level to logical objects mapped from physical elements, thus expedite the procedure of design and programming compared with node-based languages.

Virtual machines such as Mate [51] and SensorWare [11] allow large sensor networks to be reprogrammable frequently by writing application scripts, replicating them through the network and executing them automatically. However, they usually concentrate on issues related to code replication and auto-execution rather than raising programming abstraction levels. For example, to reduce energy cost of code replication, Mate even provides an instruction-like language to shorten code length, which actually puts extra burden on programmers shoulders.

Database-centric abstractions such as TinyDB [64][65][66] and Cougar [99] view sensor networks as databases that allow users to express requirements as queries, and to distribute and execute these queries.

Comparatively, our work, instead of providing a specific data collection and aggregation model, attempts to support a wider range of applications by encapsulating not only computation and communication units but also actuation units into its programming abstractions.

Event-based models such as [55] are similar with database-centric abstractions except that they view the sensor field as an active entity that automatically push data streams to users when defined events are triggered instead of a passive database which only responds upon queries.

Group-based primitives such as Hood [95] and Abstract Regions [92], provide neighbor discovery and neighborhood data sharing mechanisms. Compared with EnviroSuite, these abstractions are passive. In contrast, EnviroSuite abstractions are active objects that encapsulate local code and aggregate state, as well as share data across neighborhoods or regions.

Another group-based paradigm, State-centric programming [56], described a programming abstraction mostly related to our work. However, it is implemented and evaluated only on Pieces simulator built in Java and Matlab, which can not simulate some critical features of wireless communication including message collision. In contrast, our work includes a detailed implementation in nesC on TinyOS, an operating system for real sensor network devices, and provides comprehensive evaluation results both in TOSSIM and real sensor devices. Furthermore, the underlying group management protocol [58] differs in its mechanisms for object classification and identity management.

Finally, we should mention that a criticism of current high-level programming languages has been that they are too application specific. Hence, intermediate-level languages such as [73] were proposed as a step towards macroprogramming. EnviroSuite attempts to cater to a general application pool by diversifying the supported object types.

6.2 Data Storage

Most storage services were usually designed for individual nodes, such as ELF [17], Matchbox [26], MicroHash [100], and Capsule [69]. Several distributed storage services have also been designed [18][24][84]. One example is TSAR [18], which features a two-tier storage and indexing architecture: local storage at the sensor nodes and distributed indexing at the proxies. TSAR is different from our design in that storage is not a cooperative activity among nodes. DIMENSIONS [24] is another system that is designed to store long-term information by constructing summaries at different spatial resolutions using various compression techniques. TinyDB [64][65][66] and related projects [10][99] organize sensor networks and their collected data as a distributed database and focus on query processing techniques to acquire data from such databases. All these services assume connected operation and real-time data acquisition. Geared for disconnected operation, EnviroStore has a completely different focus. Namely, it investigates cooperation between different tiers (sensor nodes, data mules [78], and basestations) to maximize storage capacity.

One key challenge of our distributed storage service is load balancing. Load balancing has been used for other purposes in sensor networks, including maximizing system lifetime by balancing energy consumption of different nodes [54], and improving fairness by balancing MAC layer accesses [97]. Load balancing in EnviroStore is similar to the former, where the available storage space is similar to the remaining battery for each node. However, previous energy load-balancing algorithms can not be directly used for EnviroMic because when applied to storage, we have the additional control knob of exchanging data between nodes.

More broadly, load-balancing comprises many algorithms that are studied in different application contexts. Representative applications include load-balancing in web servers [96][102], P2P networks [82][49], wireless LANs [7], and distributed operations systems [14]. These applications commonly involve many nodes, which can range from web servers to P2P clients, each with a finite resource capacity. The particular resource may be bandwidth, computing power, or storage space. When more resources than desired are consumed, a node tries to reduce its resource consumption by transferring some load to its peers. While this general description also applies to EnviroStore, EnviroStore is considerably different. First, EnviroStore has the extra constraint of limited energy, which leads to new insights such as lazy offload. Second, because of the resource limitations of individual nodes, no single node is able to coordinate with all the other nodes. Load balancing in EnviroStore must be completely distributed, dependent only on local information. Third, EnviroStore has the additional challenge of redistributing data between entities that are disconnected.

6.3 Debugging Tools

In recent years, sensor network researchers have proposed several tools and middleware that aid the debugging and evaluation of sensor network applications. Generally, they can be divided into four categories: simulators, emulators, test-beds and services. The section compares EnviroLog with related work in each of these categories.

Simulators are popular tools in debugging and evaluation of sensor network applications since they don't usually require the deployment of sensor hardware. NS-2 [1], GloMoSim [101] and TOSSIM [52] are good examples. NS-2 is a discrete event simulator supporting various networking protocols over wired and wireless networks. GloMoSim focuses more on mobile, wireless networks. It allows comparison of multiple protocols at a given layer. TOSSIM is a simulator especially designed for TinyOS applications, which provides scalable simulations of sensor network software. Current simulators, however, do not adequately capture the real behavior of sensor networks. This is due to the difficulty in modeling practical imperfections such as radio irregularity as well as due to the lack of good models of environmental inputs. The ability of EnviroLog to record environmental events can presumably be utilized to improve these tools by importing recorded event data to simulate environmental inputs.

Another category of debugging and performance evaluation tools in sensor networks is emulators that mimic sensor devices either in software or hardware. AVR JTAG ICE [4], a real time in-circuit emulator, is a good representative of hardware emulators. It uses the JTAG interface to enable a user to do real-time emulation of the microcontroller of sensor devices. A drawback of such in-circuit emulators is that they have to be physically connected to emulated devices, which causes logistical difficulties in conducting experiments especially for large-scale applications covering a wide field. Atemu [75] is a software emulator for AVR-processor-based systems that emulates AVR processors as well as other peripheral devices on the MICA2 platform. Like TOSSIM, Atemu also simulates wireless communication. Such software emulators do not introduce the logistical difficulties exhibited in hardware emulators, but they are usually less realistic in reproducing network behavior.

The final stages of debugging and performance tuning typically use actual testbeds to evaluate sensor network applications. For example, Motelab [94] is a public testbed using MICA2 platforms, which allows users to upload executables and receive execution results via the Internet. Kansei [86] is another testbed. It employs XSM, MICA2, and Stargate platforms. EmStar [28] is a combination of emulators and testbeds for Linux-based sensor network applications, which runs applications using either a modeled radio channel or the channel of real nodes. EmTOS [29] extends EmStar to run TinyOS applications by compiling them into EmStar binaries. These testbeds ease the development and evaluation a lot without requiring full-scale deployment. However, they do not focus on repeatability of environmental inputs like EnviroLog does.

We categorize all other software facilitating field tests of sensor network applications as services. EnviroLog belongs to this category. Monitoring tools such as Message Center [87] aid field tests by capturing messages in the air, filtering and displaying them to users. Closest to EnviroLog is TOSHILT [47], a middleware for hardware-in-the-loop testing. TOSHILT defines emulated stimuli to replace the real environmental events, so that applications can be evaluated repeatedly before the final deployment. Since TOSHILT uses synthetic and parametric event profiles, the detail of accuracy is less than what can be captured by EnviroLog. In addition, TOSHILT doesn't provide abstractions similar to EnviroLog annotations to ease the integration of the middleware into user applications. All these difference make EnviroLog unique. In the following, EnviroLog is described in more detail.

6.4 Target Tracking

Target tracking has received special attention in recent ad hoc and sensor networks literature. Many prior approaches (e.g., in the ubiquitous computing and communication domains) focused on tracking cooperative targets. Cooperative targets are those that allow themselves to be tracked typically by exporting a unique identifier to the infrastructure (such as a cell-phone number). Examples of cooperative targets include cell phones, RFID tags [81] and smart badges [91]. Since such devices are preconfigured with a unique identity, the tracking problem is generally reduced to that of locating the uniquely identified device and performing hand-offs if needed (e.g., in cellular phones). In contrast, we focus on tracking non-cooperative targets such as enemy vehicles that do not broadcast self-identifying information. The presence and identity of such targets can only be inferred from sensory signatures, as opposed to direct communication with the target. Tracking non-cooperative targets is more challenging due to the difficulty in associating sensory signatures with the corresponding targets (e.g., all tanks look the same to our unsophisticated motes). The presence of target mobility further complicates the tracking problem.

One of the first research efforts on group management for non-cooperative target tracking has been conducted by researchers at PARC [57][58]. Their group management method dynamically organizes sensors into collaborative groups, each of which tracks a single target. Typical tracking problems such as multi-target tracking and tracking crossing targets are solved elegantly in a distributed way. Other approaches to target tracking include [3] which presents a particle filtering style algorithm for tracking using a network of binary sensors which only detect whether the object is moving towards or away from the sensor. A scalable distributed algorithm for computing and maintaining multi-target identity information in described in [79]. In [103] a tree-based approach is proposed to facilitate sensor node collaboration in tracking a mobile target.

We have investigated the tracking problem in several of our publications. Similar to [57], in [9] we present a set of group management algorithms which form sensor groups at the locations of environmental events of interest and attach logical identities to the groups. Based on [9], EnviroTrack [2] proposes an environmental computing paradigm which facilitates tracking application development. EnviroSuite [61] further extends the paradigm to support a broader set of applications that are not limited to target tracking. Geared for tracking of fast-moving targets with low communication cost on available hardware platforms that have limited sensing and communication abilities, we proposed several group management optimizations in [60] for EnviroSuite. Such optimizations may be applicable to other systems such as [58] as well.

6.5 Acoustic Applications

Sensor network applications have used acoustic sensors for different purposes, including localization [80], surveillance [32], communication [88], and geophysical monitoring [93]. Interestingly, none of these applications let users retrieve raw acoustic sensor samplings: they either use filtered samplings for application needs [32], or use acoustic signals for purposes other than recording [80]. Comparatively, in EnviroMic, we store raw sampling results in a cooperative manner into local storage (flash memory), and retrieve them later upon user request. The way we handle data remotely echoes data-mule [78], which also uses a store-and-fetch model.

For acoustic systems, challenges arise to handle the high data volume generated by high frequency sampling of acoustic sensors. To tackle the problem, EnviroMic mainly focus on reducing data redundancy. Obviously, other techniques including in-network filters [30] and data compression algorithms [76] can be easily integrated into EnviroMic to further reduce the data volume to be stored in network.

Also, EnviroMic has a great potential to be applied in applications which previously did not use acoustic sensors. For example, in animal monitoring [67][59] EnviroMic may characterize the behavior of animals from perspectives totally different from previous approaches using temperature or GPS sensors: acoustic data are much richer in nature and provide direct reflections of animal behavior. Furthermore, data retrieved by EnviroMic can be correlated with data from other sensors to reveal hidden behavior patterns that may not be possible to obtain without using acoustic sensors. The authors of [43] implemented an acoustic sensor network application that monitors cane toads. However, they assume the existence of more storage-rich devices for real-time data uploading.

Chapter 7

Conclusions

It is envisioned that sensor network technology is to bring about the next society-transforming change since the Internet, leading to a new computing, sensing and actuation infrastructure that is deeply embedded into our environment, represents the environment in bits, and ultimately interconnects human beings and the surrounding physical world.

As initial steps towards this goal, this dissertation focuses on creating usable development support for the community. The goal is to simplify programming and reduce development cost. We proposed and designed one of the first high-level programming paradigms in sensor networks. We also implemented and empirically evaluated a library of middleware services to support the programming paradigm. As a whole, our efforts constitute a comprehensive programming framework that enables rapid development of sensor network applications.

Inside the framework, we included object-based abstractions (EIP) and language primitives to simplify design and implementation of sensor network systems. Our abstractions capture the unique feature of sensor systems, their rich interactions with the physical environment, by seamlessly combining objects in the logical world with physical elements in the external environment. Programming of sensor network systems that monitor and track physical activities is thus made much more intuitive and less time-consuming.

We provided a library of middleware services to support the abstractions during runtime. Central to the library are a set of object management protocols that maintain the unique mapping between logical and physical objects. Using both simulations and experiments on real hardware platforms, we demonstrated the effectiveness of such protocols. These object management services define new, simple ways to locally collect aggregate states of environmental elements.

A secondary set of middleware services manage and retrieve the collected data. Rather than focusing on how to retrieve data in real-time via multi-hop communication, we concentrated our effort on data storage and retrieval services for delay-tolerant sensor networks. Simulation results show that the cooperative storage service we developed achieved orders of magnitude improvement in terms of prolonging lifetime of in-network storage.

The EIP abstractions and the support library simplify the design and implementation phases of sensor network application development. Before finally deploying the applications, programmers have to conduct the last phase, which is to extensively test the applications under realistic environments to fine-tune and verify their performance. Therefore, we also investigated testing and debugging solutions in order to reduce the cost of in-field experiments. The recording and replay tool we developed resolved the problem of non-repeatability of asynchronous events originated from the dynamically changing environment. A thorough measurement of overhead and performance of the replay service validates its ability to achieve repeatability with fairly low overhead.

In this dissertation, we put special emphasis on practicability of the developed framework. We provided not only fine-grained measurements of the overhead and performance of individual protocols and services via simulations and small testbed experiments, but also extensive in-field evaluation of several integrated systems built upon the framework. Empirical results collected from realistic deployments of such systems validate the efficacy of our framework in building practical systems.

This dissertation extensively explored the problem of programming for a single, isolated sensor network. Along with the same line, we are still in search for innovative solutions and techniques to address the programming issues in sensor networks that are globally interconnected.

Increased miniaturization and reduced cost of hardware (by Moore's law) will soon lead to the proliferation of embedded sensing devices (e.g., motes, RFIDs, cell-phones, cameras, GPS in cars, etc.). There are growing interests in globally interconnecting such devices so that relevant information sources can be combined to improve the quality of decision making. With the addition of myriads of new sensing devices to the network, the total bandwidth of data production can grow unboundedly. In contrast, the human capacity for data consumption does not evolve as quickly. Neither our cognitive abilities (to consume input) nor our population changes fast enough to match the increase in net data production offered by the explosion of new embedded sensing devices. In order to avoid the human bottleneck, Computing devices will have to become more autonomous with progressively less human mediation. These trends herald the emergence of omnipresent, autonomous, globally connected networks of embedded sensing devices with their own meanings of interaction with the physical environment.

An explosion of applications built upon such globally interconnected sensor networks, expanding way beyond current temperature collecting and tank tracking systems, is expected to occur soon. There are multiple application directions to be explored:

- In the medical domain, the retirement of the baby boom generation will place an enormous burden on the current healthcare system. Interconnected smart homes, as one important application of sensor networks, will help the generation live more independently, which makes the working generation more productive as to sustain the US economy.
- In the Internet domain, the increasing complexity of computing and the associated infrastructure, middleware, and applications will soon reach a level beyond human ability to control and manage. There is an increasing need for (performance) sensors and means to automate collection and processing of information to regulate themselves to deliver high-quality service, to collaboratively diagnose and cure temporary failures for 24/7, and to lessen the increasing demands for skilled administrators.
- In the business world, globalization of the economy gives rise to huge international companies with globally distributed warehouses. RFID-based sensor networks can certainly be applied to ease the complex inventory control.

The emergence of globally connected sensor networks and the explosion of application directions offer a variety new challenges to address the deficiencies of current solutions.

Our framework addressed the programming of a single network of homogeneous sensor devices. Yet, there is little support for writing applications that extend beyond a single sensor network patch, employ heterogeneous hardware platforms, and operate within multiple autonomous domains. In the future, we are interested in addressing this issue by designing and developing a programming framework and network architecture that allows developers to easily implement large-scale systems operating on multiple sensor network patches interconnected via the Internet. This architecture should seamlessly integrate low-level abstractions (e.g., for domain experts to develop sophisticated signal processing and data fusion algorithms) and high-level abstractions (e.g., for application developers to quickly implement decision and control systems that use high-level information such as the outputs of data fusion). The same architecture should support diverse applications from smart homes to autonomic server management, and to many others.

Due to the vast number of sensors deployed, most sensor systems will be left unattended for prolonged periods. Failures in hardware or software might not expose themselves until events of interest occur, which may severely harm system performance (imaging a system that failed to detect a severe heart attack or failed to send alarms for defected power stations or damaged power lines). Online validation is of great importance for such systems. Test cases containing external inputs and expected outputs should be fed into the system so that they can be executed on demand or periodically to validate whether the system is performing appropriately. We plan to continue our initial effort in EnviroLog to generate repeatable external inputs as well as develop general ways for diverse sensor network systems to express validation procedures.

Advances in hardware miniaturization and wireless techniques will soon make the vision of seamlessly embedded, omnipresent, self-organized, globally connected networks of embedded sensing devices become reality. It is our aspiration that our current and future research may contribute to expedite the realization of the vision and benefit the lives of millions.

References

[1] The network simulator - ns-2, 2006. `http://www.isi.edu/nsnam/ns/`.

[2] T. Abdelzaher, B. Blum, Q. Cao, D. Evans, J. George, S. George, T. He, L. Luo, S. Son, R. Stoleru, J. Stankovic, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *ICDCS '04: Proceedings of the International Conference on Distributed Computing Systems*, 2004.

[3] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus. Tracking a moving object with a binary sensor network. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 150–161, New York, NY, USA, 2003. ACM Press.

[4] Atmel Corporation. Mature avr jtag ice.
`http://www.atmel.com/dyn/products/tools_card.asp?tool_id=2737`.

[5] S. G. Azevedo and T. E. McEwan. Micropower impulse radar. *Science and Technology Review*, 1996.

[6] M. A. Batalin, M. Rahimi, Y. Yu, D. Liu, A. Kansal, G. S. Sukhatme, W. J. Kaiser, M. Hansen, G. J. Pottie, M. Srivastava, and D. Estrin. Call and response: experiments in sampling the environment. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 25–38, New York, NY, USA, 2004. ACM Press.

[7] G. Bejerano and S.-J. Han. Cell breathing techniques for balancing the access point load in wireless lans. In *Infocom '06: Proceedings of the 25th Conference on Computer Communications*, 2006.

[8] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Trans. Comput. Syst.*, 2(1):39–59, 1984.

[9] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 201–214, New York, NY, USA, 2003. ACM Press.

[10] P. Bonnet, J. Gehrke, and P. Seshadri. Towards sensor database systems. In *MDM '01: Proceedings of the Second International Conference on Mobile Data Management*, pages 3–14, London, UK, 2001. Springer-Verlag.

[11] A. Boulis, C.-C. Han, and M. B. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *MobiSys '03: Proceedings of the 1st international conference on Mobile systems, applications and services*, pages 187–200, New York, NY, USA, 2003. ACM Press.

[12] E. Cheong, J. Liebman, J. Liu, and F. Zhao. Tinygals: a programming model for event-driven embedded systems. In *SAC '03: Proceedings of the 2003 ACM symposium on Applied computing*, pages 698–704, New York, NY, USA, 2003. ACM Press.

[13] E. Cheong and J. Liu. galsc: A language for event-driven embedded systems. In *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pages 1050–1055, Washington, DC, USA, 2005. IEEE Computer Society.

[14] M. Correa, A. Zorzo, and R. Scheer. Operating system multilevel load balancing. In *SAC '06: Proceedings of the 2006 ACM symposium on Applied computing*, pages 1467–1471, New York, NY, USA, 2006. ACM Press.

[15] Crossbow. Mica motes, 2007. `http://www.xbow.com`.

[16] Crossbow Technology Inc. MTS300 Multi Sensor Board, 2006. `http://www.xbow.com`.

[17] H. Dai, M. Neufeld, and R. Han. Elf: an efficient log-structured flash file system for micro sensor nodes. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 176–187, New York, NY, USA, 2004. ACM Press.

[18] P. Desnoyers, D. Ganesan, and P. Shenoy. Tsar: a two tier sensor storage architecture using interval skip graphs. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 39–50, New York, NY, USA, 2005. ACM Press.

[19] 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 (LCN'04)*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[20] A. Dunkels, O. Schmidt, T. Voigt, and M. Ali. Protothreads: simplifying event-driven programming of memory-constrained embedded systems. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 29–42, New York, NY, USA, 2006. ACM Press.

[21] P. Dutta, M. Grimmer, A. Arora, S. Bibyk, and D. Culler. Design of a wireless sensor network platform for detecting rare, random, and ephemeral events. In *IPSN '05: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks*, 2005.

[22] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *MobiCom '99: Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, pages 263–270, New York, NY, USA, 1999. ACM Press.

[23] K. Fall. A delay-tolerant network architecture for challenged internets. In *SIGCOMM '03: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 27–34, New York, NY, USA, 2003. ACM Press.

[24] D. Ganesan, B. Greenstein, D. Perelyubskiy, D. Estrin, and J. Heidemann. An evaluation of multi-resolution storage for sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 89–102, 2003.

[25] R. K. Ganti, P. Jayachandran, T. F. Abdelzaher, and J. A. Stankovic. Satire: a software architecture for smart attire. In *MobiSys 2006: Proceedings of the 4th international conference on Mobile systems, applications and services*, pages 110–123, 2006.

[26] D. Gay. Matchbox: A simple filing system for motes, 2003. `http://www.tinyos.net/tinyos-1.x/doc/matchbox.pdf`.

[27] 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 Press.

[28] L. Girod, J. Elson, A. Cerpa, T. Stathopoulos, N. Ramanathan, and D. Estrin. Emstar: a software environment for developing and deploying wireless sensor networks. In *Proceedings of the 2004 USENIX Technical Conference*, June 2004.

[29] L. Girod, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys'04)*, November 2004.

[30] B. Greenstein, C. Mar, A. Pesterev, S. Farshchi, E. Kohler, J. Judy, and D. Estrin. Capturing high-frequency phenomena using a bandwidth-limited sensor network. In *SenSys '06: Proceedings of the 4th ACM Conference on Embedded Networked Sensor Systems*, November 2006.

[31] L. Gu, D. Jia, P. Vicaire, T. Yan, L. Luo, A. Tirumala, Q. Cao, T. He, J. A. Stankovic, T. Abdelzaher, and B. H. Krogh. Lightweight detection and classification for wireless sensor networks in realistic environments. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 205–217, New York, NY, USA, 2005. ACM Press.

[32] L. Gu, D. Jia, P. Vicaire, T. Yan, L. Luo, A. Tirumala, Q. Cao, T. He, J. A. Stankovic, T. Abdelzaher, and B. H. Krogh. Lightweight detection and classification for wireless sensor networks in realistic environments. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, 2005.

[33] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *DCoSS*, 2005.

[34] R. Gummadi, N. Kothari, T. Millstein, and R. Govindan. Declarative failure recovery for sensor networks. In *AOSD '07: Proceedings of the 6th international conference on Aspect-oriented software development*, pages 173–184, New York, NY, USA, 2007. ACM Press.

[35] 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 Press.

[36] T. He, S. Krishnamurthy, L. Luo, T. Yan, L. Gu, R. Stoleru, G. Zhou, Q. Cao, P. Vicaire, J. A. Stankovic, T. F. Abdelzaher, J. Hui, and B. Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, 2006.

[37] T. He, S. Krishnamurthy, J. A. Stankovic, T. Abdelzaher, L. Luo, R. Stoleru, T. Yan, L. Gu, J. Hui, and B. Krogh. Energy-efficient surveillance system using wireless sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 270–283, New York, NY, USA, 2004. ACM Press.

[38] J. Heo, D. Henriksson, and T. Abdelzaher. Autonomous energy optimization in multi-tier server farms. In *In submission*, 2007.

[39] J. Heo, L. Luo, T. Abdelzaher, and A. Kulhari. Mstp: A multiple-source transport layer abstraction for sensor networks. In *In submission*, 2007.

[40] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *ASPLOS-IX: Proceedings of the ninth international conference on Architectural support for programming languages and operating systems*, pages 93–104, New York, NY, USA, 2000. ACM Press.

[41] J. L. Hill and D. E. Culler. Mica: A wireless platform for deeply embedded networks. *IEEE Micro*, 22(6):12–24, 2002.

[42] M. Horton, D. Culler, K. PIster, J. Hill, R. Szewczyk, and A. Woo. Mica: The commercialization of microsensor motes. *Sensors*, 19(4):40–48, April 2002.

[43] W. Hu, V. N. Tran, N. Bulusu, C. T. Chou, S. Jha, and A. Taylor. The design and evaluation of a hybrid sensor network for cane-toad monitoring. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 71, 2005.

[44] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *MobiCom '00: Proceedings of the 6th annual international conference on Mobile computing and networking*, pages 56–67, New York, NY, USA, 2000. ACM Press.

[45] Intel. Intel mote, 2007. `http://www.intel.com/research/exploratory/motes.htm`.

[46] S. Jain, K. Fall, and R. Patra. Routing in a delay tolerant network. In *SIGCOMM '04: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 145–158, New York, NY, USA, 2004. ACM Press.

[47] D. Jia, B. H. Krogh, J. Cole, and P. Dolan. Toshilt: Middleware for hardware-in-the-loop testing of wireless sensor networks, 2006. `http://www.ece.cmu.edu/w̃ebk/sensor_networks`.

[48] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. S. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: design tradeoffs and early experiences with zebranet. *SIGOPS Oper. Syst. Rev.*, 36(5):96–107, 2002.

[49] K. Kenthapadi and G. S. Manku. Decentralized algorithms using both local and random probes for p2p load balancing. In *SPAA '05: Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 135–144, New York, NY, USA, 2005. ACM Press.

[50] M. M. H. Khan, L. Luo, C. Huang, and T. Abdelzaher. Snts: Sensor network troubleshooting suite. In *DCOSS '07: Proceedings of the 2005 International Conference on Distributed Computing in Sensor Systems*, June 2007.

[51] P. Levis and D. Culler. Mate: 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*, 2002.

[52] P. Levis, N. Lee, M. Welsh, and D. Culler. Tossim: accurate and scalable simulation of entire tinyos applications. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 126–137, New York, NY, USA, 2003. ACM Press.

[53] P. Levis, S. Madden, D. Gay, J. Polastre, R. Szewczyk, A. Woo, E. Brewer, and D. Culler. The emergence of networking abstractions and techniques in tinyos. In *NSDI '04: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.

[54] Q. Li, J. Aslam, and D. Rus. Online power-aware routing in wireless ad-hoc networks. In *Mobicom*, 2001.

[55] S. Li, Y. Lin, S. H. Son, J. Stankovic, and Y. Wei. Event detection services using data service middleware in distributed sensor networks. *Telecommunication Systems, Special Issue on Information Processing in Sensor Networks*, 26(2-4), 2004.

[56] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. State-centric programming for sensor-actuator network systems. *Pervasive Computing, IEEE*, 2(4):50–62, 2003.

[57] J. Liu, M. Chu, J. Liu, J. Reich, and F. Zhao. Distributed state representation for tracking problems in sensor networks. In *IPSN '04: Proceedings of the 3rd International Symposium on Information Processing in Sensor Networks*, pages 234–242, April 2004.

[58] J. Liu, J. Liu, J. Reich, P. Cheung, and F. Zhao. Distributed group management for track initiaition and maintenance in target localization applications. In *IPSN '03: Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks*, April 2003.

[59] T. Liu, C. M. Sadler, P. Zhang, and M. Martonosi. Implementing software on resource-constrained mobile sensors: experiences with impala and zebranet. In *MobiSYS '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 256–269, New York, NY, USA, 2004. ACM Press.

[60] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Design and comparison of lightweight group management strategies in envirosuite. In *DCOSS '05: Proceedings of the 2005 International Conference on Distributed Computing in Sensor Systems*, July 2005.

[61] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic. Envirosuite: An environmentally immersive programming framework for sensor networks. *Trans. on Embedded Computing Sys.*, 5(3):543–576, 2006.

[62] L. Luo, T. He, G. Zhou, L. Gu, T. F. Abdelzaher, and J. A. Stankovic. Achieving repeatability of asynchronous events in wireless sensor networks with envirolog. In *Infocom '06: Proceedings of the 25th Annual IEEE Conference on Computer Communications*, April 2006.

[63] L. Luo, C. Huang, T. F. Abdelzaher, and J. A. Stankovic. Envirostore: A cooperative storage system for disconnected operation in sensor networks. In *Infocom '07: Proceedings of the 26th Annual IEEE Conference on Computer Communications*, May 2007.

[64] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tag: a tiny aggregation service for ad-hoc sensor networks. *SIGOPS Oper. Syst. Rev.*, 36(SI):131–146, 2002.

[65] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. The design of an acquisitional query processor for sensor networks. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 491–502, New York, NY, USA, 2003. ACM Press.

[66] S. R. Madden, M. Franklin, J. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1), March 2005.

[67] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson. Wireless sensor networks for habitat monitoring. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 88–97, New York, NY, USA, 2002. ACM Press.

[68] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, New York, NY, USA, 2004. ACM Press.

[69] G. Mathur, P. Desnoyers, D. Ganesan, and P. Shenoy. Capsule: An energy-optimized object storage system for memory-constrained sensor devices. In *SenSys '06: Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems*, November 2006.

[70] M. Melkonian. Get by without an rtos. *Embedded Systems Programming*, September 2000.

[71] Microsoft. Ole2 programmers reference, 1994.

[72] Moteiv. Tmote, 2007. `http://www.moteiv.com`.

[73] R. Newton, Arvind, and M. Welsh. Building up to macroprogramming: An intermediate language for sensor networks. In *IPSN '05: Proceedings of the Fourth International Conference on Information Processing in Sensor Networks*, 2005.

[74] Ohio State University. Xsm motes, 2007. `http://www.cast.cse.ohio-state.edu/exscal`.

[75] J. Polley, D. Blazakis, J. McGee, D. Rusk, J. S. Baras, and M. Karir. Atemu: A fine-grained sensor network simulator. In *SECON '04: the First IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, October 2004.

[76] C. M. Sadler and M. Martonosi. Data compression algorithms for energy-constrained devices in delay tolerant networks. In *SenSys '06: Proceedings of the Fourth ACM Conference on Embedded Networked Sensor Systems*, November 2006.

[77] R. Shah, S. Roy, S. Jain, and W. Brunette. Data mules: Modeling a three-tier architecture for sparse sensor networks. In *SNPA*, May 2003.

[78] R. C. Shah, S. Roy, S. Jain, and W. Brunette. Data mules: Modeling a three-tier architecture for sparse sensor networks. In *IEEE Workshop on Sensor Network Protocols and Applications (SNPA)*, 2003.

[79] J. Shin, L. Guibas, and F. Zhao. A distributed algorithm for managing multi-target identities in wireless ad-hoc sensor networks. In *IPSN '03: Proceedings of the 2nd International Workshop on Information Processing in Sensor Networks*, April 2003.

[80] G. Simon, M. Maroti, A. Ledeczi, G. Balogh, B. Kusy, A. Nadas, G. Pap, J. Sallai, and K. Frampton. Sensor network-based countersniper system. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 1–12, New York, NY, USA, 2004. ACM Press.

[81] H. Stockman. Communication by means of reflected power. pages 1196–1204, October 1948.

[82] S. Surana, B. Godfrey, K. Lakshminarayanan, R. Karp, and I. Stoica. Load balancing in dynamic structured peer-to-peer systems. *Perform. Eval.*, 63(3):217–240, 2006.

[83] R. Szewczyk, A. Mainwaring, J. Polastre, J. Anderson, and D. Culler. An analysis of a large scale habitat monitoring application. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 214–226, New York, NY, USA, 2004. ACM Press.

[84] S. Tilak and N. B.Abu-Ghazaleh. Collaborative storage management in sensor networks. In *International Journal on Ad Hoc and Ubiquitous Computing, Vol 1, Nos. 1/2*, 2005.

[85] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 51–63, New York, NY, USA, 2005. ACM Press.

[86] O. S. University. *Kansei: Sensor Testbed for At-Scale Experiments*, Feb 2005.

[87] Vanderbilt University. Message Center. *http://www.isis.vanderbilt.edu/projects/nest/msgctr.html*.

[88] I. Vasilescu, K. Kotay, D. Rus, P. Corke, and M. Dunbabin. Data collection, storage, and retrieval with an underwater sensor network. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, 2005.

[89] S. Vinoski. Corba: Integrating diverse applications within distributed heterogeneous environments. *IEEE Communications Magazine*, 32(2):46–55, February 1997.

[90] R. von Behren, J. Condit, and E. Brewer. Why events are a bad idea for high-concurrency servers. 2003.

[91] R. Want, A. Hopper, V. Falco, and J. Gibbons. The active badge location system. *ACM Trans. Inf. Syst.*, 10(1):91–102, 1992.

[92] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *NSDI '04: Proceedings of the First USENIX/ACM Symposium on Networked Systems Design and Implementation*, March 2004.

[93] G. Werner-Allen, K. Lorincz, J. Johnson, J. Lees, and M. Welsh. Fidelity and yield in a volcano monitoring sensor network. In *OSDI '06: Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation*, November 2006.

[94] G. Werner-Allen, P. Swieskowski, and M. Welsh. Motelab: A wireless sensor network testbed. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks (IPSN'05), Special Track on Platform Tools and Design Methods for Network Embedded Sensors (SPOTS)*, April 2005.

[95] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *MobiSys '04: Proceedings of the 2nd international conference on Mobile systems, applications, and services*, pages 99–110, New York, NY, USA, 2004. ACM Press.

[96] J. L. Wolf and P. S. Yu. Load balancing for clustered web farms. *SIGMETRICS Perform. Eval. Rev.*, 28(4):11–13, 2001.

[97] A. Woo and D. E. Culler. A transmission control scheme for media access in sensor networks. In *MOBICOM*, 2001.

[98] N. Xu, S. Rangwala, K. K. Chintalapudi, D. Ganesan, A. Broad, R. Govindan, and D. Estrin. A wireless sensor network for structural monitoring. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 13–24, New York, NY, USA, 2004. ACM Press.

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

[100] D. Zeinalipour-Yazti, S. Lin, V. Kalogeraki, D. Gunopulos, and W. A. Najjar. Microhash: An efficient index structure for flash-based sensor devices. In *FAST '05: Proceedings of the 4th USENIX Conference on File and Storage Technologies*, December 2005.

[101] X. Zeng, R. Bagrodia, and M. Gerla. Glomosim: a library for parallel simulation of large-scale wireless networks. In *PADS '98: Proceedings of the twelfth workshop on Parallel and distributed simulation*, pages 154–161, Washington, DC, USA, 1998. IEEE Computer Society.

[102] Q. Zhang and W. Sun. Workload-aware load balancing for clustered web servers. *IEEE Trans. Parallel Distrib. Syst.*, 16(3):219–233, 2005. Member-Alma Riska and Member-Evgenia Smirni and Senior Member-Gianfranco Ciardo.

[103] W. Zhang and G. Cao. Optimizing tree reconfiguration for mobile target tracking in sensor networks. In *Infocom '04: Proceedings of the 23rd Conference of the IEEE Communications Society*, March 2004.

Author's Biography

Liqian Luo was born and grew up in Zhejiang province, China. In 1996, she left for Beijing and entered the Department of Computer Science and Technology at Tsinghua University. She received her Bachelor's degree from Tsinghua University in 2000. Two years later, she joined the University of Virginia where she was advised by Professor Tarek Abdelzaher. She received the degree of Master of Computer Science from the University of Virginia in 2004. She continued her Ph.D. study at the University of Illinois at Urbana-Champaign from 2005 to 2007.

Liqian's primary research interests are in the areas of sensor networks, distributed embedded systems, wireless/mobile communication, and ubiquitous computing. She is the developer of EnviroSuite, a sensor networks programming language that implements Environmentally-Immersive Programming. She is also one of the primary developers of VigilNet, an integrated sensor network system for surveillance applications that was later classified and transitioned to the Defense Intelligence Agency (DIA). She is a Vodafone Graduate Fellow and the recipient of the UIUC Distinguished Networking and Systems Student Award.