A SERVICE-ORIENTED ARCHITECTURE FOR DYNAMIC MACROPROGRAMMING OF SENSOR NETWORKS

BY

KIRILL ALEXANDROVICH MECHITOV

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, 2011

Urbana, Illinois

Doctoral Committee:

Professor Gul Agha, Chair & Director of Research
Professor Bill Spencer
Associate Professor Tarek Abdelzaher
Associate Professor Indranil Gupta

Abstract

In the late 1990s, advances in sensing and computer technology have enabled the development of tiny, inexpensive, low-power wireless sensor platforms. By integrating sensing, communication, and computational capabilities, these *smart sensors* were poised to revolutionize our view of the environment we inhabit by linking the physical world with the digital realm of traditional computing. Smart sensors have been available to researchers for more than a decade; however, few large-scale applications have emerged outside the laboratory setting, and the commercial potential of this technology has been limited. The principal reason for this outcome is the difficulty inherent in programming wireless sensor networks (WSNs) consisting of more than a handful of sensors: built from inexpensive components, individual nodes in this distributed system are prone to failures; interaction with the physical world imposes real-time constraints on computation and communication; and the limited energy of battery-powered sensor nodes leads to stringent energy efficiency requirements. Combined, these challenges have caused WSN software development to lag behind the capabilities offered by the hardware.

The goal of this research is to enable robust, large-scale application development for wireless sensor networks, allowing the full potential of WSN technology to be realized. To this end, we leverage two powerful techniques, *service-oriented architecture* (SOA) and *macroprogramming*. Adapting SOA, which is typically seen in Internet-scale web applications, to WSNs enables application components to cooperate and share limited resources in an intelligent manner, while providing useful high-level programming abstractions to the application developer. Macroprogramming—specifying the aggregate behavior of a distributed system rather than each node individually—builds on SOA to create lightweight, mobile applications that can combine and control the services resident in the network to take advantage the capabilities of the network as a whole.

This approach has proven successful, enabling a long-term deployment of a dense array of structural health monitoring (SHM) sensors on a cable-stayed bridge in Jindo, South Korea. The software resulting from this work, which integrates the service-oriented application development framework with a suite of domain services and comprehensive applications for SHM, has been released as the open-source Illinois SHM Services Toolsuite. It is currently in use by over 70 research groups worldwide.

To my parents, Alex Mechitov and Helen Moshkovich.

Acknowledgments

I am deeply grateful to my advisor Gul Agha for his guidance and support during my studies. His deep knowledge, insight, and experience have been an invaluable resource in the course of my research. Likewise, I am in debt to Professor Bill Spencer, with whom I have collaborated extensively on structural health monitoring. His enthusiastic support has been an important motivation. I thank Professors Tarek Abdelzaher and Indranil Gupta for their critical evaluation and refinement of this research, as well as their support and encouragement. I am also grateful to Professors Chung-Bang Yun and Hyung-Jo Jung of KAIST for their help and support during my stays in Korea.

I would like to express special gratitude to my colleagues for extensive discussions and collaboration, without which my work would have been impossible. I am especially grateful to Reza Razavi, with whom I worked UIUC and the University of Luxembourg to develop the core macroprogramming ideas. YoungMin Kwon and I worked together for many years on issues of localization and mobile agents, and his research provided valuable complement to my work. Tomonori Nagayama and Jennifer Rice have been a great help with all things related to structural health monitoring. I am grateful for our frequent discussions and fruitful collaboration. I also appreciate the work on various aspects of sensor networks done jointly with Wooyoung Kim, Lauren Linderman, Parya Moinzadeh, and Sameer Sundresh. Robin Kim has my thanks for relentless testing and bug hunting, which have contributed greatly to the success of the ISHMP software.

During my time at UIUC, I have met and worked with many exceptional people in the Open Systems Laboratory and the Smart Structures Technology Laboratory. I would like to thank them for their friendship, help, and advice: Tom Brown, Jingo Chen, Liping Chen, Bill Donkervoet, MyungJoo Ham, Nadeem Jamali, Myeong-Wuk Jang, Shinae Jang, Hongki Jo, Rajesh Karmani, Vijay Korthikanti, Jian Li, Sherin Moussa, Mehwish Nagda, Koushik Sen, Reza Shiftehfar, Sung-Han Sim, Prasanna Thati, Predrag Tosic, and Reza Ziaei. I am also grateful to Soojin Cho, Jongwoong Park, and the rest of the KAIST team with whom I worked during the Jindo Bridge deployments.

Finally, I would like to thank the friends I made over the years in Champaign-Urbana, and my family for their patience and enduring support.

Table of Contents

Chapter 1

Introduction

Wireless sensor networks (WSNs) represent the merger of the physical world, its processes and environments, with the information infrastructure constantly growing around us as computing resources become truly pervasive. WSNs are a crucial part of the *cyber-physical systems* (CPS) environment, which is a large-scale distributed system comprising a mix of low-power embedded computing devices, sensing and actuation elements, networked mobile devices, and traditional computing and network platforms.

One of the principal challenges of computer science research in cyber-physical systems is to find ways of creating scalable, robust, and efficient software capable of operating in this environment. Programming WSNs is particularly challenging due to the lack of sophisticated software engineering tools and programming languages commonly used in modern large-scale software development. Due to resource constraints and efficiency requirements, low-level C programming remains the dominant application development method in this domain.

The goal of this research is to enable robust, large-scale application development for wireless sensor networks, allowing the full potential of WSN technology to be realized and made accessible to a wider group of users. Computer scientists and systems programmers are *not* the intended users of such systems, and with large-scale adoption of ubiquitous WSN technology still many years in the future, nor is the public at large. Rather, scientists and engineers are the people who will actually program these systems to gather, process, and act on information provided by the sensors.

We approach this problem through a combination of *macroprogramming*, a technique that makes use of high-level programming languages to specify the global behavior of the system rather than the behaviors of its individual components, with *service-oriented architecture*, a scalable, adaptive software architecture model typically used for Internet-scale web applications, and a mobile agent-based execution framework. The result, which we call *dynamic macroprogramming*, enables on the fly specification, deployment and execution of WSN tasks by application programmers, in a robust and resource-efficient manner. This work is guided by the requirements of a real-world WSN system, and has culminated in the development of an open source service-oriented framework for building applications to monitor the condition of civil infrastructure.

1.1 Motivation

The concept of *Ambient Intelligence* envisions the "invisible" incorporation into our surrounding environment and everyday objects of billions of loosely-coupled sensing, actuating, computing and communicating components as part of an intelligent ambient infrastructure. The aim is to find new ways of supporting and improving people's lives—ways that go far above and beyond what is possible with personal computers today—by creating *ambient software systems* that exploit this infrastructure and enable new work practices in many fields including scientific observation and experimentation, monitoring (environment, health, industrial process), control, etc. Cyber-physical systems, which combine digital computation with sensing and actuation of the physical world, are one constituent of this ambient infrastructure.

CPS components such as microelectromechanical systems (MEMS), smart materials, "smart sensor" networks, and bio-inspired software are subject to intensive research and development. Another key technology is short-range, low-power *ad hoc* wireless networking of potentially very large numbers of autonomous, distributed, and low-powered computers endowed with limited processing, memory, sensing, actuation, and communication capabilities. Wireless sensor networks are prime examples of this paradigm, focusing on sensing, data acquisition, and information synthesis.

Current practice considers wireless sensor networks in the context of a single application, e.g., a WSN for target tracking or a WSN for environment monitoring. Large-scale deployments (over 100 sensors) are as yet uncommon. This model of application development, together with the small scale of most experimental sensor network deployments, has led to the design of WSN protocols and services that is highly efficient, yet often static, tightly coupled or customized to a particular application or protocol. This practice hampers code portability and reuse, and presents a high barrier to entry for the development of novel WSN applications.

Some recent research has addressed supporting several concurrent tasks on a sensor network [92]. As sensor deployments become more numerous and their scale increases, we see sensor networks transitioning from being a *part* of the application to becoming an *ambient computing platform* used for executing multiple tasks concurrently. For instance, as illustrated in Figure 1.1, middleware and application-level services can be shared among unrelated coexisting applications [58]. In this context, efficient low-level code customized to a particular application is a poor solution, as common functionality is needlessly replicated across applications, and coordination is lacking.

When a WSN is viewed in this manner, it becomes clear that situations will arise where multiple end-users would want to exploit this ambient computing infrastructure on their own, without calling in embedded systems programmers, by executing existing programs, or modifying and composing their own high-level programs, or macros, on the fly. This is motivated by the diversity of functionalities that may be required

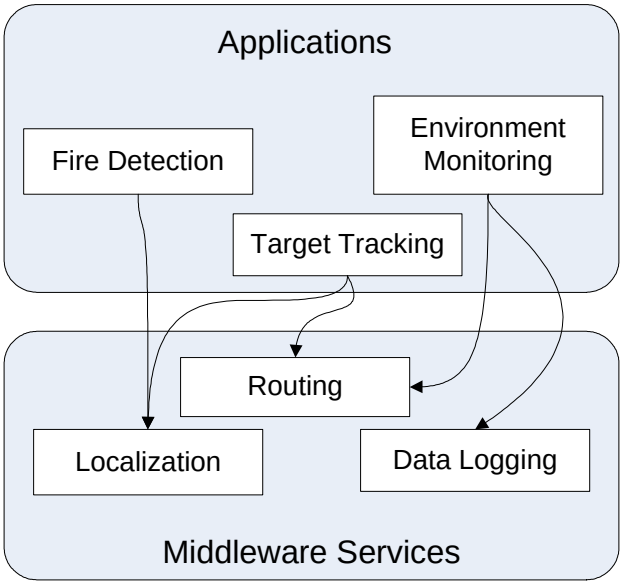


Figure 1.1: Many-to-many relationship between applications and middleware services.

from this system, further amplified by the unpredictability of the phenomena being monitored and the potential changes in the ambient computing infrastructure.

Currently, software development tools and flexible software architectures suitable to this open, dynamic WSN environment are lacking. In particular, programming WSNs is exceptionally hard due to their resource limitations. Moreover, sensor nodes are prone to failure (for example, if the energy supply runs out), and low-power wireless communication between them is unreliable. Programming such systems requires addressing these limitations. Unfortunately, current methods for WSN programming lead developers to mix high-level concerns such as quality of service requirements, for instance timeliness, reliability, application logic, adaptivity, with low-level concerns like resource management, synchronization, communication, routing, data filtering and aggregation. This makes developing software for WSNs a costly and error-prone endeavor, even for expert programmers.

The experience of domain experts (scientists and engineers) trying to use WSNs in their work offers supporting evidence that current WSN programming practices are untenable, and a more robust approach is required. For example, Nagayama and Spencer employed smart sensor networks for structural health monitoring (SHM) of civil infrastructure [62]. This research resulted in the successful development of a laboratory-scale WSN for bridge monitoring; however, it soon became apparent that the implementation was cumbersome to modify or transition to deployments on structures with different parameters from the original design. A more modular approach, based on a composition of middleware services, has been pursued as a result [64, 78].

1.2 Problem Statement

We consider cyber-physical systems comprising both WSNs as well as more traditional computing artifacts such as desktop and server PCs, embedded computers, and handheld mobile devices. Let us call each hardware component of this system an *ambient node*. Ambient systems are open in that both ambient nodes and application tasks using them can be added and removed dynamically. Based on resource availability and optimization criteria, available ambient nodes coordinate and decide at runtime their mutual application execution responsibilities.

Running on this ambient computing infrastructure are cyber-physical applications that make use of a number of general middleware services, such as routing, localization, and time synchronization. Additionally, the applications themselves may contain service-like reusable components, e.g., signal detection or target tracking algorithms, gradient search, data compression, and others. In order to accommodate the vast collection of services and protocols already developed by the sensor network community, we adopt a very broad definition of a middleware service, concerning ourselves only with their interfaces to applications and other services, and not their internal semantics or implementation methods.

Existing system development and deployment techniques do not appear satisfactory for programming such ambient systems. New computation models and software development methodologies are required. Satoh observes, for instance, that "Ambient intelligence technologies are expected to combine concepts of intelligent systems, perceptual technologies, and ubiquitous computing" [80].

In our view, these requirements imply the need for a software architecture that provides a *looser coupling* between services and applications, and among the services themselves, in a resource-efficient and context-aware manner: a dynamic service composition-based architecture for WSN systems, with the dual goals of facilitating large-scale software development and enabling global, network-wide optimization, rather than application-focused local optimization. Dissociating middleware services from the application context and from each other gives up some possible performance advantages due to explicit customization and tight coupling. In return, a more scalable software development process, support for multiple concurrent tasks, and the possibility of global resource management across applications are gained.

Macroprogramming has been proposed as a technique for facilitating WSN programming. According to Mainland et al., "macroprogramming enables the specification of a given distributed computation as a single global specification that abstracts away low-level details of the distributed implementation" [54]. This makes macroprograms more accessible to application programmers. It then becomes the task of the programming environment to compile this high-level specification down to executable low-level operations that are implemented by individual ambient nodes, and then deploy and carry out these operations [71, 75].

We call the combination of these techniques—dynamic service-oriented software architecture and user-driven macroprogramming—*dynamic macroprogramming*. The following list of requirements emerges for dynamic macroprogramming of WSNs:

- Program representation in terms familiar to domain experts, who may not have extensive embedded systems programming experience
- Expressive, high level system-wide program specification environment
- Concurrent evaluation and execution of the resultant macroprograms in a shared WSN environment, with real-time feedback to the programmer
- Dynamic run-time code deployment and execution on an open ambient computing platform, in a resource-efficient manner

1.3 Contributions

The contributions of this thesis are as follows. First, we present the design of two novel techniques for WSN software development: dynamic macroprogramming and service-oriented architecture. Combined, this programming framework presents a powerful, flexible programming environment for wireless sensor networks. Second, we develop a two-level implementation of the dynamic macroprogramming framework composed of interacting meta-actors and mobile agents executing service requests in the network. This approach enables lightweight coordination and network-wide resource optimization between multiple services and applications. ActorNet, an actor language specifically designed for wireless sensor network systems, provides the runtime platform for this system. To complement the macroprogramming environment, we develop a suite of robust, energy-efficient middleware services that are critical most WSN tasks. Finally, we complete a full-scale experimental validation of the comprehensive programming approach via a long-term deployment of over 100 sensors for structural health monitoring. The software created in the course of this research, including the middleware services and SHM algorithms, is publicly available as part of the open-source Illinois SHM Services Toolsuite [36].

The remainder of the thesis is organized as follows. First, we present some background and related research material in Chapter 2. Next, we introduce the design of service-oriented architecture and dynamic macroprogramming for sensor networks in Chapters 3 and 4. Chapter 5 describes the ActorNet programming language, which provides the foundation of the dynamic macroprogramming architecture. The following chapters examine the design and implementation of critical middleware services, including synchronized

sensing, ranging, localization, and two canonical WSN applications taking advantage of these services: object tracking and structural health monitoring. We conclude with the description of a full-scale deployment of a wireless sensor network powered by the Illinois SHM Services Toolsuite, providing experimental validation of the software development framework, in Chapter 9 and a brief discussion of the results and future work in Chapter 10.

Chapter 2

Background

2.1 Wireless Sensor Networks and Cyber-Physical Systems

The essential difference between a standard sensor and *smart sensor* is the latter's flexible information processing capability. Smart sensors have an on-board microprocessor that can be used for digital signal processing, self-diagnostics, self-identification and self-adaptation functions. Furthermore, all smart sensor platforms developed thus far have employed wireless technology. Wireless sensor networks, a collection of smart sensors acting in concert, are perhaps the most common constituent of cyber-physical systems currently being investigated. A National Research Council report [66] noted that the use of networked systems of embedded computers and sensors throughout society could well dwarf all previous milestones in the information revolution.

Numerous researchers have developed smart sensing platforms; Lynch and Loh [53] cited over 150 papers on wireless sensor networks conducted at over 50 research institutes worldwide. These platforms can be grouped into three primary categories: 1) proprietary, 2) open hardware/software, and 3) proprietary hardware with open source software and interfaces. Many proprietary platforms have been developed by individual research groups. While this research has advanced the state-of-the-art in smart sensor technology, in general progress has been slow due to the lack of coordination and leveraging.

A turning point in the development of smart sensor technology was the release of the "mote" from UC Berkeley [32]. Under the substantial support of the US Defense Advanced Research Projects Agency (DARPA), a general-purpose, open hardware/software platform was developed and made available to the research community. The first mote, termed COTS Dust, incorporated communications, processing, sensors, and batteries into a package about a cubic inch in size. Subsequently, the Rene Mote was released in 2000 and the Mica Mote was released in 2001 [31]. To date, Berkeley Motes (Mica, Mica2, Mica Dot, MicaZ, Telos) have tailored mainly to low power, low data rate applications, requiring minimum data processing.

Intel has developed the Imote2 platform (see Figure 2.1), which like the Berkeley Mote runs TinyOS [2]. The Imote2 provides enhanced computation and communication resources that allow demanding data-intensive

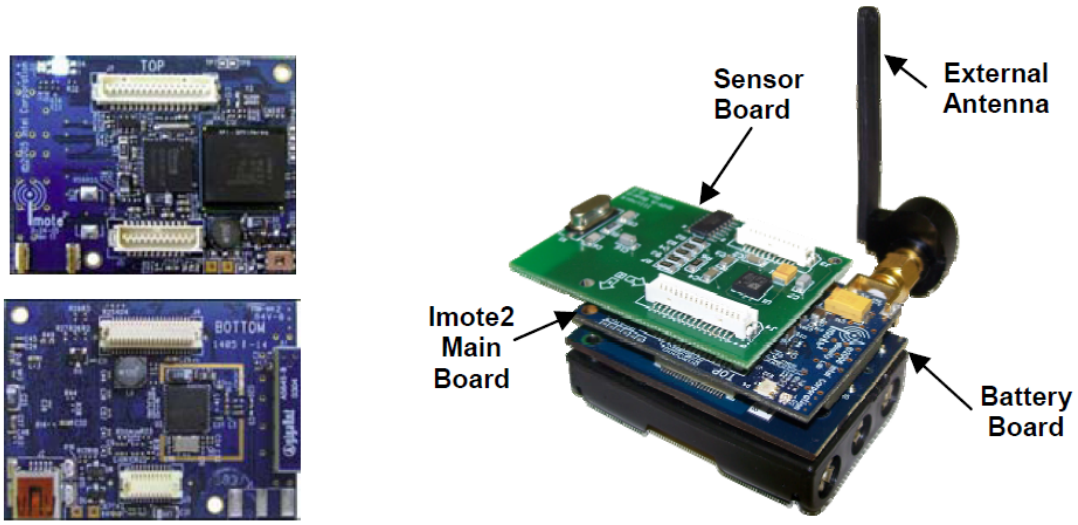


Figure 2.1: Top and bottom of Imote2 main board (left) and stackable configuration (right).

sensor network applications to be supported. The processor speed may be scaled based on the application demands, thereby improving its power efficiency. One of the important characteristics of the platform, which separates it from other commercially available wireless sensing nodes, is the memory size. The Imote2 has 256 KB of integrated SRAM, 32 MB of external SDRAM, and 32 MB of Flash memory.

Sensor nodes are typically battery-powered, and the limited supply of energy has been an important limitation on the longevity of sensor networks and their usefulness for long-term unattended monitoring applications. Power harvesting techniques have been explored as a means to prolong the lifetimes of wireless embedded devices [73]. Power harvesting electronics generate energy from environmental sources such as solar energy or ambient vibrations.

2.2 Service-Oriented Architecture and Macroprogramming

Service-oriented architecture (SOA) has recently been proposed as a way to bring this design philosophy to building dynamic, heterogeneous distributed applications spanning the Internet [83, 87]. Different applications can be built from the same set of services depending on how they are linked and on the execution context [28]. This approach makes for dynamic, highly adaptive applications without the need to revisit and adapt the implementation of each service for a particular application context.

SOA has been applied to address the inherent problems in designing complex and dynamic WSN applications [51, 58, 75]. Building an application from a set of well-defined services moves much of the complexity associated with embedded distributed computing to the underlying middleware.

A survey of solutions currently proposed in the literature reveals a variety of approaches to macroprogramming WSNs: EnviroSuite [52], LiteOS [10], and Semantic Streams [90] are widely cited. Although many of these approaches are quite powerful, none of them provide the language abstractions required for dynamic macroprogramming by end-users.

EnviroSuite proposes environmentally immersive programming, an object-based programming model in which individual objects represent physical elements in the external environment. In both EnviroSuite and ActorNet [45], actors or objects must be created explicitly by programmers to provide a service. Behavioral specifications are not in terms of groups of agents. Protocols to support operations over groups of objects and protocols to implement such specifications may not be re-used.

The SONGS architecture and programming model considers sensor network applications as a composition of semantic services [51]. Semantic services are a type of semantic data transformation functions, and do not correspond exactly to what we call services in our work. We are interested in facilitating composition of less structured application and middleware services, a vast quantity of which has already been developed for wireless sensor networks.

Compilation-based approaches such as the ATaG data-driven macroprogramming language [71] and the Regiment macroprogramming system [67] have also been proposed. These rely on models or *a priori* knowledge of the system environment to generate executable code. In contrast, dynamic macroprogramming requires real-time monitoring of the system to provide up to date contextual information needed to enable a high degree of adaptivity.

Other comparable approaches include the Tenet architecture for tiered sensor networks [27], Sensor Webs [16], and Sensor Grids [50]. A major difference these systems possess compared to a dynamic macroprogramming architecture is that we do not attribute *a priori* master and slave roles to the architecture components, masters being traditional computing elements having the responsibility to control the behavior of slave nodes. In addition, none of these architectures provides the combination of an expressive program specification with a Turing-complete mobile code deployment and execution environment. Further, network flooding techniques are in general used for dynamic code deployment, instead of fine-grained code deployment available in a mobile agent platform.

Finally, Levis et al. [48] observe that a fairly complicated actions, such as transmitting a message over the radio, could be represented as a single bytecode instruction provided by an application-specific instruction set, and provides a framework for implementing high-level application-specific virtual machines on motes and for disseminating bytecode. By their nature, such application-specific virtual machines are inappropriate in environments where multiple uncoordinated applications may coexist.

2.3 Agents and Mobile Code

Some form of mobile code deployment is required to implement dynamic macroprogramming. Traditionally, WSN application development involved fairly static programming languages, operating systems and reprogramming services, for efficiency reasons. For example in TinyOS [32], the sensor network application components are written in NesC [25], a C-derivative language, and compiled together with the operating system code and middleware services into a single application image. The image can then be uploaded to the sensor nodes using the Deluge protocol [35] prior to program execution. This approach proves successful in achieving its stated goal of highly efficient utilization of sparse computing resources. Unfortunately, it is ill-suited for an open system comprising a dynamic set of diverse, transient tasks that is the expected workload in ambient systems.

The Melete system provides a method for concurrently executing uncoordinated applications in a sensor network [92]. Melete applications are written in the TinyScript language and executed by a virtual machine on an arbitrary subset of nodes in the network. We propose a more comprehensive method of executing concurrent applications in cyber-physical systems, which allows greater dynamism in the application and the physical environment, context-awareness and adaptivity and a higher level of optimization [58]. In fact, Melete may be used as part of our architecture, acting as the code deployment method for service instances.

ActorNet [45] is a mobile agent platform for WSNs, designed to support multiagent applications on these resource-limited, real-time systems. ActorNet agents, are called actors, and are based on the actor model of computation: concurrent active objects communicating via asynchronous message passing. The concept of the actors was proposed by Hewitt [30], and formalized as a transition system by Agha [3]. There are many implementations of Actor systems, including the work of Agha et al. [4], where the location of an actor computation is added to the actor programs to enhance the concurrency. ActorNet actors are specified in a high-level interpreted language based on Scheme, which allows highly dynamic, mobile code to be executed across multiple sensor nodes. We employ ActorNet as the code deployment framework for dynamic macroprogramming, since Scheme-like code is very easy to generate automatically based on a behavior template.

Agilla [23] is another WSN mobile agent platform, and is in many respects similar to ActorNet. The principal difference is in the trade-off between power and expressiveness of the actor language versus efficiency. Agilla agents are based on virtual machine code, which is considerably more compact than ActorNet's Scheme representation. For the same reason, however, Agilla agents are not as flexible or capable as their ActorNet counterparts.

2.4 Time Synchronization

The problem of time synchronization is well known and has been studied extensively in the context of distributed systems. Several algorithms have been proposed to address a variety of issues that arise in time synchronization, including precision, efficiency, reliability and fault tolerance. In particular, efficient algorithms exist for synchronizing clocks in systems with arbitrary clock rate variance [70] and under various failure models [47]. Likewise, time synchronization in wireless sensor networks has been widely investigated. Taking advantage of broadcast communication, sensors can assess relative differences among the local clocks in their immediate neighborhood. Reference Broadcast Synchronization [21] and Flooding Time Synchronization Protocol [56] are among the well-known synchronization methods.

Clock rate difference (i.e., drift) from node to node increases time synchronization error over time. When clock drift is significant, synchronization is applied periodically before the error becomes too large. Mechitov et al. implemented FTSP on Mica2 motes as a part of wireless data acquisition system for structural health monitoring [57]. This system can maintain better than 1 millisecond synchronization accuracy for extended periods without the need for resynchronization. Thus, fine-grained synchronization among wireless sensor nodes has been shown to be achievable.

However, measured signals may not be synchronized with each other, even when clocks are precisely synchronized. Individual sample timing is not necessarily controlled precisely based on these clocks. Synchronized sensing has been employed in a large-scale sensor network for monitoring the Golden Gate Bridge [42]. The approach relied on a *real-time operating system* (RTOS) and a hardware platform that allows for fine-grained control of sample acquisition times. In contrast, the solution we propose does not rely on RTOS extensions or specific properties of the sensor platform.

2.5 Ranging and Localization

The Global Positioning System (GPS) is by far the most popular standard for electronic outdoor localization [33]. However, GPS units are either too costly or too imprecise for wireless sensor networks comprising hundreds of nodes. GPS localization offers $\pm 6.3\,\mathrm{m}$ error with 95% confidence when selective availability is turned off, compared to less than 1 m error desired in many sensor network applications, such as target tracking or intrusion detection.

The Ad-hoc Positioning System (APS) is a family of distributed localization algorithms based on trilateration [68]. The basic idea is to perform multi-hop propagation of distances to anchors throughout the network, so that every node can trilaterate its position. Four different distance metrics were developed,

ranging from minimum hop count and sum of hop lengths—for isotropic, uniform density networks—to local geometric constructions, which require higher anchor density to control error propagation.

Another approach is *Multidimensional Scaling* (MDS), which has been proposed as the basis for a centralized robust localization algorithm given a set of distance measurements [15]. MDS is a technique that can be applied to calculate positions of nodes given a set of distances. One problem with this centralized approach is that it requires distances between all pairs of nodes. Distance measurement errors are one of the primary causes of localization errors, and have a great impact on the overall localization accuracy. Zhang et al. characterize common types of measurement errors and propose a method for filtering out internally-inconsistent measurements [93].

The localization problem has also been approached from a theoretical perspective. It has been proven that in the general case, localization with noisy measurements is NP-complete [6]. However, a polynomial-time algorithm based on semidefinite programming has been shown to solve a more constrained version of the localization problem [84].

2.6 Object Tracking

Mobile object tracking is a canonical problem in signal processing research and in sensor networks in particular. As such, a plethora of solutions to this and related problems have been proposed. We briefly review some research that is particularly relevant to tracking objects in outdoor environments using WSNs.

A string of research on tracking targets using WSNs originated from the DARPA SensIT program. Among the noteworthy results are a tracking framework for WSNs using geographical information [9, 49] and information utility-based target tracking [94]. The former takes a traditional approach to target tracking: it dynamically divides the region of interest based on the target's velocity and tracks multiple targets simultaneously by classifying them [49] and associating each with a particular track [9]. The latter approach attempts to select the next sensor node that most likely results in "the greatest benefit at the lowest cost" for estimation.

Among the research results on target tracking for binary sensor networks, the most closely related to our work is the method of Aslam et al. [5]. It employs a network of sensors that report whether an object is moving toward or away from them and applies a particle filtering based algorithm using geometric properties unique to the system. Reliable sensor outputs and rather long sensing ranges are assumed for the simulation-only evaluation, suggesting that the solution may be less resilient when deployed in real environments where sensor measurements are likely noisy and sensors have a limited detection range.

2.7 Structural Health Monitoring

Structural Health Monitoring (SHM) strategies measure structural response and aim to effectively detect, locate and assess damage produced by severe loading events and by progressive environmental deterioration [62]. We use the SHM application domain as a testbed for service-oriented architecture. SHM provides a challenging application environment, with high data rate sampling, bursty communication and periods of intensive data processing.

Many civil engineering researchers have focused on the development of SHM algorithms for estimating structural damage. Several methods have been applied to study this problem, including modal analysis methods (e.g., [79]), on-line system identification methods (e.g., [39]), and wavelet analysis (e.g., [86]). For high-fidelity damage detection, a dense array of sensors and long time histories are required. A comprehensive view of the existing SHM literature is given in [17, 85], each of which cites more than 300 papers on the subject.

Once data is acquired from the sensors, it often needs to be collected at one node for processing or exfiltration from the network. This process is called *data aggregation*. A data acquisition system for structure monitoring applications has been designed and implemented on a network of Mica2 motes, enabling centralized SHM algorithms to be used with distributed sensor networks [57]. This system provides middleware services for reliable communication, high-frequency sensing and network-wide time synchronization, which are critical for sensor network-based SHM applications. Notably, these services have been reused or adapted from a different domain (target tracking and localization) [43, 44, 59, 60].

Chapter 3

Service-Oriented Architecture

We design and implement a service-oriented architecture for a dynamic macroprogramming platform that addresses the requirements originally identified in Chapter 1. Successful realization of this novel programming paradigm for wireless sensor networks will open up ambient computing infrastructure to be exploited by a significantly wider research community. In reference to the vision of Ambient Intelligence that cyber-physical systems promise to eventually fulfill, we call this platform *Ambiance* [1].

Creating and executing applications on the Ambiance platform encompasses four stages, as illustrated in Figure 3.1:

1. *Application Presentation Environment*: application programmers compose the macroprogram in through a graphical interface that is expressed entirely in the language of the problem domain.

2. *Application Representation Environment*: programmer input is translated into a self-contained internal macroprogram representation, which we call a ubiquitous query, or *uQuery*.

3. *Application Transformation Environment*: the uQuery is transformed into executable mobile code suitable for deployment on the WSN.

4. *Application Deployment Environment*: mobile code implementing the application is scheduled, deployed and executed on the underlying system. Results are aggregated and returned to the application programmer.

Given this basic organization, we will now outline our design philosophy and resulting implementation choices for the Ambiance platform. The primary components of the system are the service-oriented application design, macroprogramming interface, dynamic service composition framework, and mobile code deployment and execution environment.

[1] Parts of this work appeared in previous publications [58, 75].

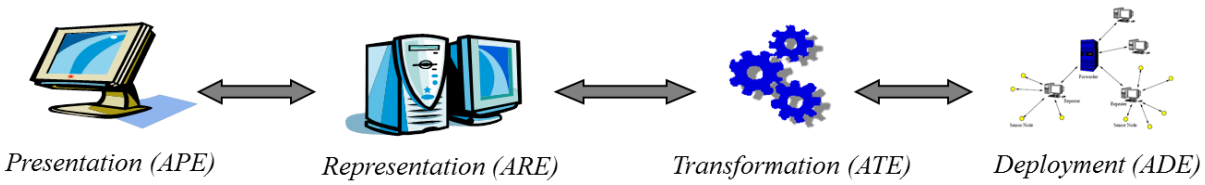


Figure 3.1: Program specification and execution process in Ambiance.

3.1 Service-Oriented Design

With the exponential growth in available computing power over the last 50 years, the complexity of computer software has likewise increased dramatically. Advances in the fields of programming language design and software engineering allow application developers to deal with this complexity by dividing the software system into smaller, manageable parts. Notably, object-oriented programming, which encapsulates data together with the methods used to operate on it, and component-based software architecture, which proposes building applications as a composition of self-contained computing components, have been instrumental to the design and development of large- scale software systems. Expanding on this idea, *service-oriented architecture* has recently been proposed as a way to bring this design philosophy to building dynamic, heterogeneous distributed applications spanning the Internet [83, 87].

Services, in SOA terminology, are self-describing software components in an open distributed system. The description of a service, called a contract, lists its inputs and outputs, explains the provided functionality, and describes non-functional aspects of execution (timeliness, resource consumption, cost, etc.). Data is passed among the services in a common format. An application consists of a composition of a number of linked services within a middleware runtime system that provides communication and coordination among them. Unlike traditional component-based architectures, services do not have to be tightly coupled with each other or operate on the same computer; indeed, the services do not have to be explicitly linked to each other until execution time. Services do not need to know who provides the input data they require, or from where it comes. Different applications can be built from the same set of services depending on how they are linked and on the execution context [28]. This approach makes for dynamic, highly adaptive applications without the need to revisit and adapt the implementation of each service for a particular application context.

SOA design principles may be applied in the sensor network context as well as on the Internet. Wireless sensor networks often consist of numerous independent nodes, each an embedded computing platform with a processor, memory, and a radio transmitter. As such, WSN applications are by definition distributed and thus require communication and coordination for parts of the application running on different nodes. SOA has been proposed to address the inherent problems in designing complex and dynamic WSN applications [51, 58].

Building an application from a set of well-defined services moves much of the complexity associated with embedded distributed computing to the underlying middleware. This approach also fosters reuse and adaptability, as services for a given application domain can be employed by a multitude of applications.

Perhaps more importantly, SOA provides for a separation of concerns in application development. That is, application designers can focus on the high-level logic of their application, service programmers can concentrate on the implementation of the services in their application domain, and systems programmers can provide middleware services (reliable communication, time synchronization, data aggregation, etc.) that enable the services to interact. In sensor networks, which at this stage are principally used by scientists and engineers, the application designer is likely to be the user of the application as well. This situation makes it especially important for the high-level design of the application and the domain-specific algorithms used by the services to be separated from the low-level infrastructure necessary to make the system work. Applying SOA to WSNs makes it possible to compose and deploy complex applications through an interface suitable for non-programmers [75]. User-driven WSN programming is a relatively young research area with few working implementations, but it holds the promise to lower the barriers to entry in sensor network application development and to make WSNs a more attractive solution.

3.2 Two-level Programming Framework

The key technical requirement of target sensor network systems is end-user programmability. This implies new techniques for on-line macroprogramming by multiple uncoordinated application programmers, coupled with the automated deployment and execution of resulting low-level code. The underlying assumption is that the users of the system belong to the large category of *domain experts*, who are described by Nardi as having the detailed task knowledge necessary for creating the knowledge-rich applications they want and the motivation to get their work done quickly, to a high standard of accuracy and completeness [65].

The architectural style of *Adaptive Object-Models* (AOM) defines a family of architectures for object-oriented software systems dynamically programmable by domain experts, and describes their key design patterns [91]. The concept of AOM was born recently from research aimed at discovering and documenting the design principles of a particular class of software with complex behavior that emerged from industry and proved most useful for processes that are rapidly changing. AOMs are *meta-level architectures* that enforce separation of concerns, and in particular the separation of high-level logic from technical aspects of implementation. They partially fulfill the requirements of dynamic macroprogramming in that non-programmer experts can customize the AOM at run-time by specifying its object-model (state and behavior),

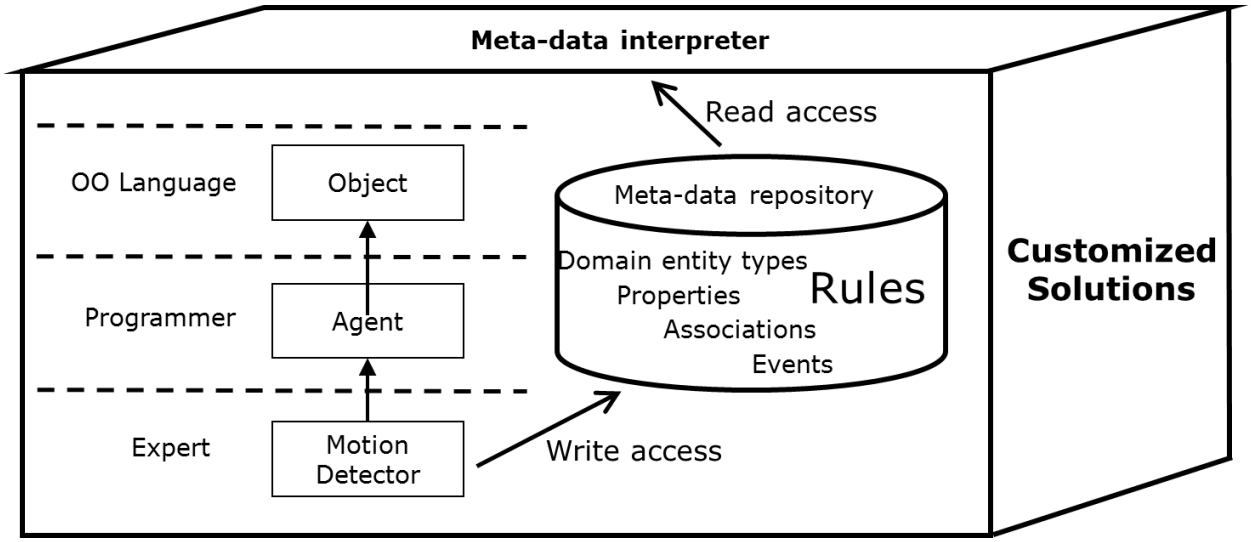


Figure 3.2: Adaptive Object-Model architectural style.

using a domain-specific language (Figure 3.2).

In effect, AOMs store the base object as used in the code alongside its meta-data description in terms accessible to the domain expert. If we consider objects as nouns, parameters as adjectives, and methods as verbs, that is the language used to describe the system to the expert. Specific knowledge of the behavior of the object in the application context is thus separated and presented to the end-user, and macroprogramming takes place entirely in this language. However, the AOM style does not provide a specific framework for representing macroprograms, and their distributed and resource-aware execution on cyber-physical systems.

We extend this architectural style with design techniques that address these issues. We enable high-level specifications of global behavior by uncoordinated end-users through a specialized Web interface, and their translation into not only meta-objects, but also meta-*actors*, which control and customize the runtime behavior of both passive and active application objects. These meta-objects are dynamic, they have the capability to observe the application objects and the environment (*introspection*), and to customize their own behavior by analyzing these observations (*intercession*), as seen in Figure 3.3.

The key innovation is the separation of the *knowledge level*, where the application, data, service definitions are represented, from the *operational level*, where actual low-level implementation of these objects and services are located and code execution takes place. Figure 3.4 provides an overview of the system decomposed into these two levels. Note that program representation and transformation environments exist entirely in the knowledge level, and are thus logically independent of the underlying execution framework used in the deployment environment.

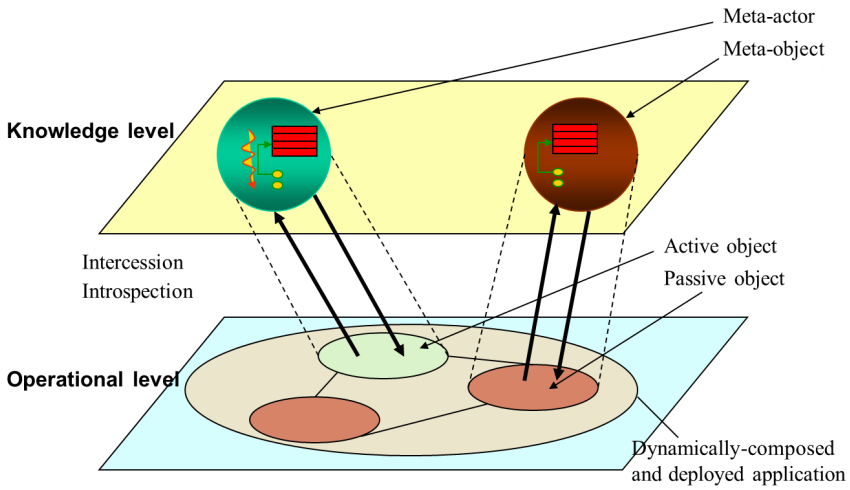


Figure 3.3: Two-level AOM architecture for controlling active objects.

3.2.1 Knowledge Level

The knowledge level keeps track of the domain knowledge and controls macroprogram processing: representing, transforming, optimizing and deploying. Specifically, it tracks two types of metadata: static metadata consisting of the domain ontology and macroprogram (uQuery) representations, and dynamic metadata consisting of the contextual information, including computed domain objects, obtained during macroprogram execution.

Dart [74], the first representation meta-model that satisfies requirements of dynamicity, end-users accessibility, as well as extendability and reusability by programmers. Figure 3.5 illustrates Dart using the UML notation. A macroprogram, called uQuery, is represented as an aggregation of *tasks*, where each task is a set of interrelated *steps*. When a step is executed, it produces a domain object. A *construct* specifies the computation method for that object. A step bridges a construct with the resulting object. The most common type of construct is the *primitive construct*, which reifies a function call. Each construct refers to, or instantiates, a *contract* which holds metadata about the construct. In the case of primitive constructs, a contract incorporates the type specification, arguments, name and a list of properties required for its execution. In order to execute a construct, we need to make platform-specific decisions, which are delegated by each contract to the associated execution strategy. The service repository for a macroprogramming engine holds these contracts. The domain ontology holds the domain concepts, together with their relationships and constraints.

Both the domain ontology and the service repository for the application domain we are interested in are assumed to be provided. These services and ontologies need to be developed separately, by domain experts

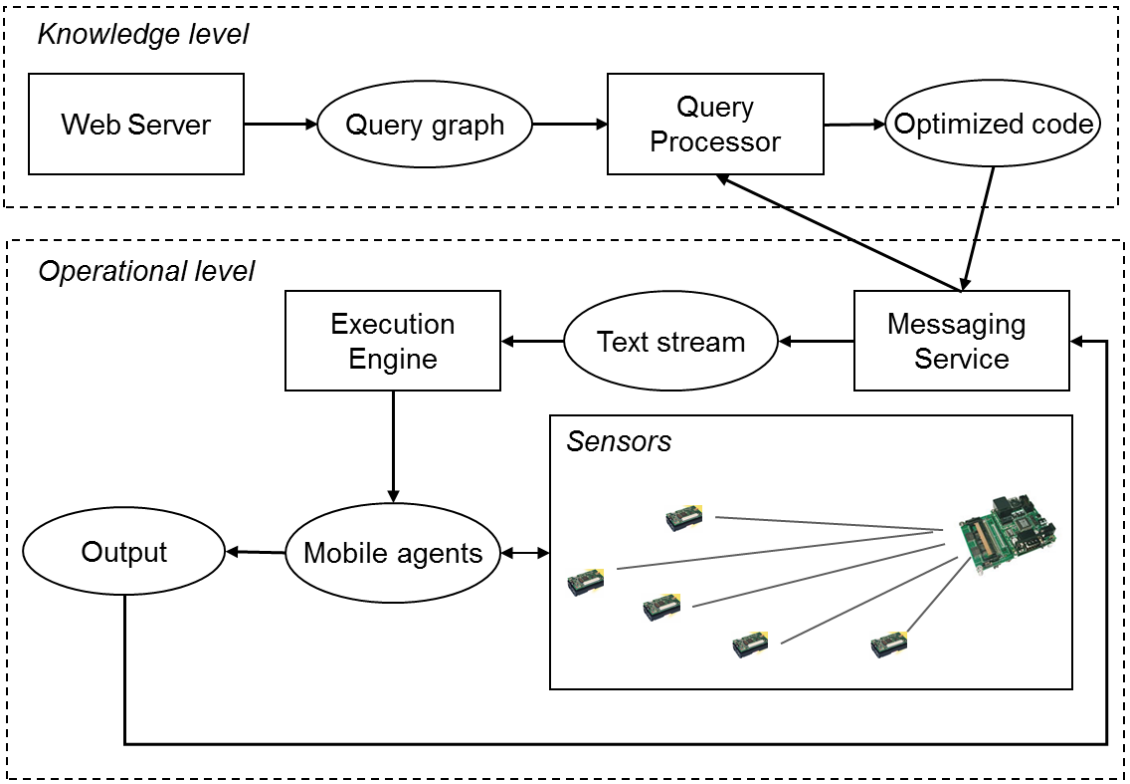


Figure 3.4: Ambiance macroprogramming platform runtime.

working alongside system programmers. We assume that the service repository is comprehensive enough to allow all macroprograms of interest to be explicitly represented as a composition of its elements.

3.2.2 Operational Level

At the operational level, a fine-grained mobile code deployment framework must be available on resource-limited, real-time distributed systems comprising the ambient infrastructure. The mobile code deployment platform is responsible for:

- Deploying and executing dynamically generated low-level code,
- Dynamically discovering and providing access to all sensor and computational resources in the system, and
- Implementing the elements of the service repository.

To the best of our knowledge, the only existing system that satisfies our operational level requirements is ActorNet [45]. ActorNet agents are called actors, based on the actor model of computation [3]. The ActorNet runtime consists of a lightweight interpreter running on each ambient node in the system, along with several

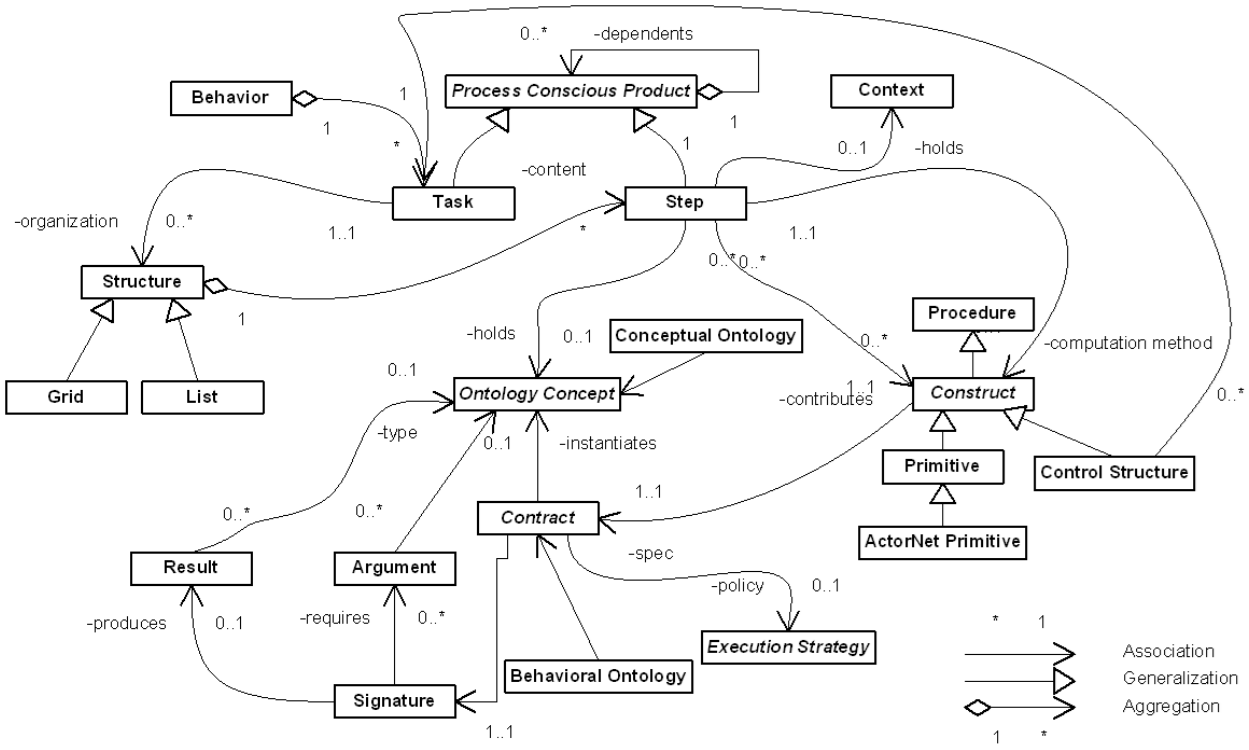


Figure 3.5: Modified Dart meta-model for representing WSN applications.

supporting services. The runtime enables the actors to execute, communicate, migrate and access sensors and actuators.

3.2.3 User Interface

A *functional specification language* is needed to create the macroprogram at the knowledge level. We employ a modified Dart framework [74] for this purpose. Dart is designed as an abstract object-oriented framework. It was initially implemented in Cincom VisualWorks Smalltalk, and used to refactor and improve the end-user programming language of a simulation system for ecology. In this case, the end-users were experts in ecology whose job was to model ecosystems (individual and social behavior of animals, plants, etc.) and to observe and study the evolution of their behavior through simulation and statistics [26].

We consider the spreadsheet model as the most appropriate basis for designing the end-user programming features of the functional specification language, since it is the most widely used end-user programming metaphor [65]. The main assumption is that complex behavior can be modeled by (1) associating operations to cells that compute objects, (2) relating together those cells as operation result holders that provide the argument to new operations. This is exactly how programming is operated through Dart-based end-user programming interfaces.

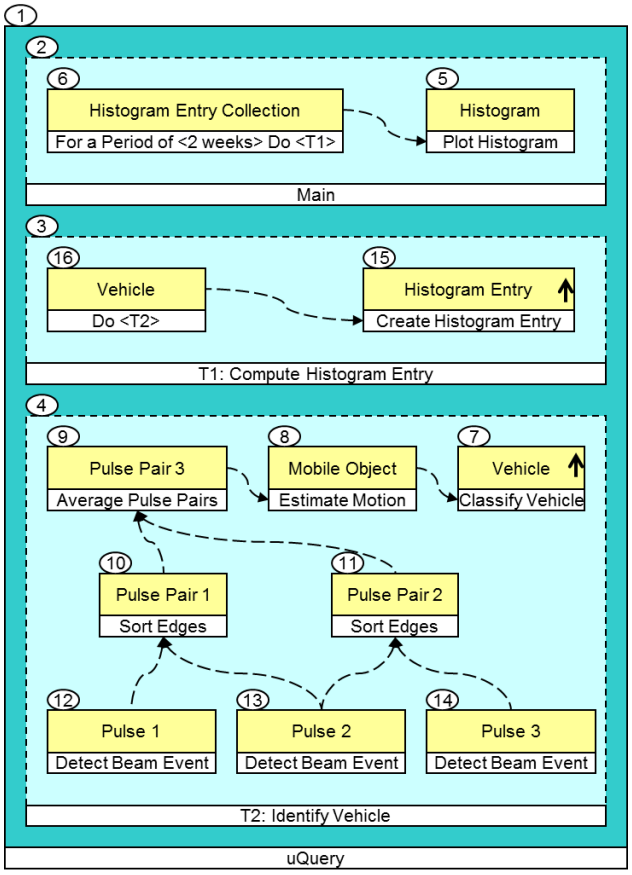


Figure 3.6: Graphical representation of an Ambiance macroprogram.

It is worthy to note that Dart extends the classic features of spreadsheets by allowing the users to:

- Integrate a set of domain and task-specific built-in primitives with relative ease.
- Associate any type of object to the spreadsheet cells, and to use that object as argument for defining new formulas (service calls).
- Incorporate user-defined functions.
- Take advantage of a rich set of general-purpose and domain-related control structures (selection, conditionals, iteration, etc.).

A web-enabled programming interface is presented to the user (actually, multiple concurrent users accessing the system over the Internet). Through this interface, programs can be composed from services and domain objects, which can be further configured and parametrized. This is akin to a bank teller using a simple fill-in form to access a client's account, without the need to know or understand the underlying database model.

Figure 3.6 presents a graphical view of a macroprogram specified in this manner, which would be the output of the web programming interface. Note that this format is used for illustration purposes only, and the actual output of the programming interface is an XML-format file containing similar information. This macroprogram, implementing the example scenario proposed originally to illustrate Semantic Streams [90], makes use of the Identify Vehicle task to assemble statistics on vehicles passing through breakbeam sensors. More realistic macroprograms would require a higher degree of concurrency, control, as well as nested or recursive tasks. All of these features are supported by the Dart framework.

3.3 Dynamic Service Composition

As the Dart framework is the basis of the user interface for our dynamic macroprogramming platform, so the dynamic service composition architecture is the basis of its execution engine.

3.3.1 Architecture Overview

Service- and component-oriented architectures are widely used, providing greater ease and scalability to the software design and implementation process. We aim to apply the same approach to the CPS domain, adapting to its unique limitations and requirements. We identify the following key principles for the design of scalable, resource-efficient CPS applications as a dynamic composition of services:

1. *System-wide programming model.* The cyber-physical system is treated as a collaborative distributed computing platform. Applications are specified as a collection of system-wide tasks and not as a unique program image per ambient node.

2. *Sharing and reuse.* Multiple uncoordinated applications and middleware services need to coexist in the network without prior knowledge of each other. Therefore, both network resources requiring exclusive access (sensors, actuators, etc.) and middleware services are shared among several applications. Resource management cannot be relegated to each application individually, it must be performed globally.

3. *Late binding.* Application specification is sufficiently flexible to allow run-time adaptivity in selecting the services and resources to be used. We do not know in advance which services or resources will be used by which application, or when. Postponing the choice of which service or resource best fits the application opens up more opportunities for optimization.

Adhering to the service-oriented architecture model of application design, we implement a service composition-based software architecture that follows from these design principles. Our architecture leverages the concept of dynamic service composition to support application development for open cyber-physical systems. It adopts the two-level architectural model, separating the two major concerns: that of controlling the execution process, including strategic decision making and adaptation, and that of the functional execution itself. First, we restate our assumptions about the problem more formally.

Assumptions

We consider applications specified in terms of a composition of calls to middleware service interfaces, and we refer to the service interface specification as a *contract* and each call to a service a *service request*. A repository of available services for a given WSN or application domain is provided.

To facilitate the use of a large number of pre-existing middleware services within our architecture, we choose not to constrain the model of a service's behavior, e.g., whether it is distributed, centralized, single-threaded, etc. Since services and applications need to interact and coordinate, however, we fix a model for their interaction. We use the Actor model of computation [3] to represent service interfaces connecting services to each other and to the application. Thus, services are used by our system as if they were implemented as *actors*: concurrent active objects interacting via asynchronous messaging. We distinguish between the actors representing the service itself from *meta-actors*, which are control entities supervising deployment and execution of the services.

Responsibilities of the meta-actor include controlling the lifecycle of a service (deploying, starting, stopping and disposing of the service) and interaction with other services. Interaction occurs through the actor interface specified in the service contract. Only interactions through actor interfaces are mediated by our architecture; any side effects are not captured by this model.

We further assume the availability of a functional service composition language, where service requests are *self-sufficient* and *minimally constrained*. The service composition language is functional in that (1) the control flow between service requests is partially ordered and driven by data dependencies, and (2) it allows for a recursive graph traversal to autonomously process each service request in the specification. Self-sufficiency refers to the fact that each individual service request is provided with the required knowledge about the arguments, resources, context and method required for its execution. Minimally constrained refers to delaying as long as possible placing constraints necessary to execute a specific instance of the service.

The last requirement is a fine-grained runtime code deployment method, such as a mobile agent system like ActorNet [45] or Agilla [23]. As long as these assumptions hold, the service composition architecture can

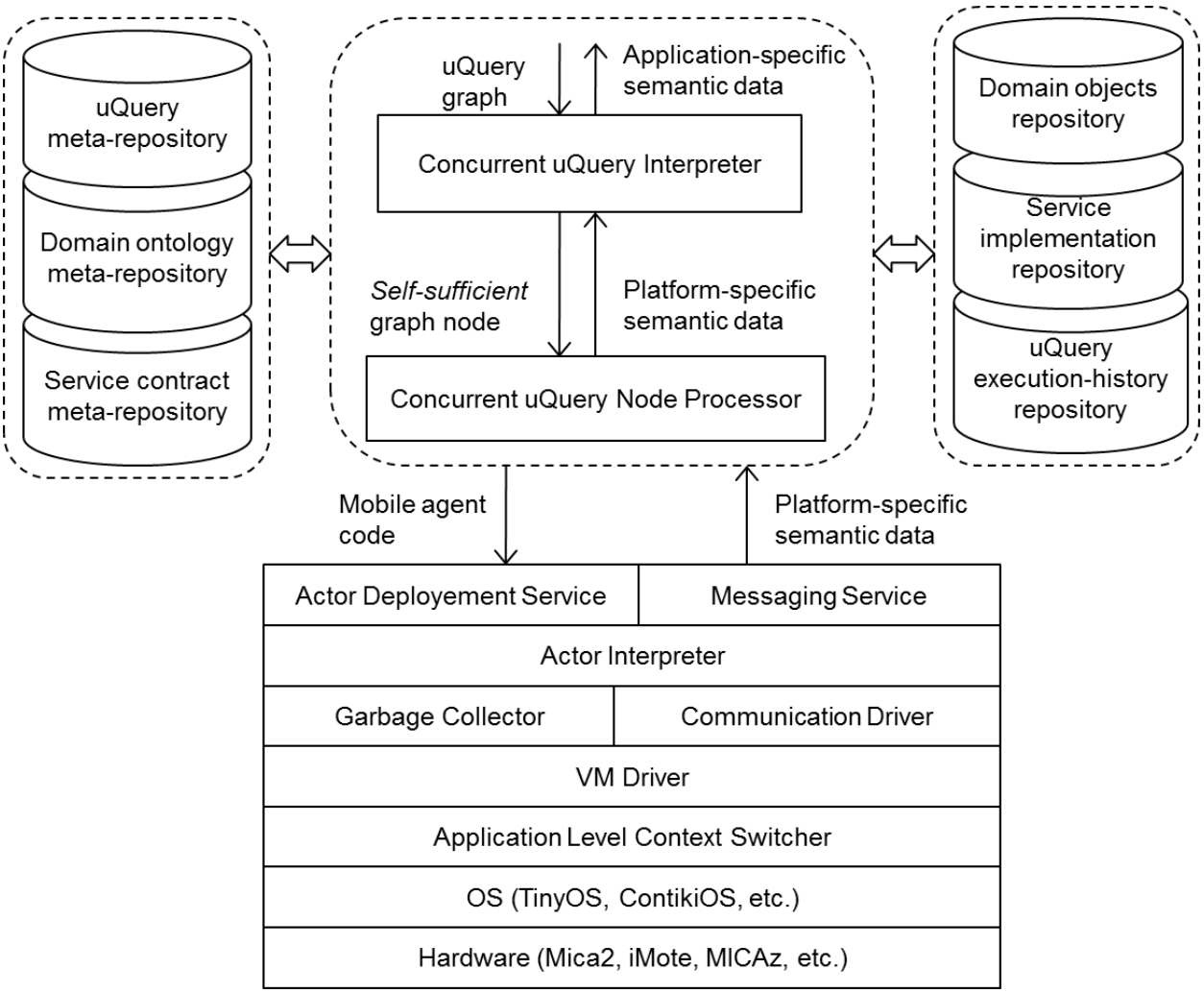


Figure 3.7: Implementation of the Ambiance dynamic service composition architecture.

be considered independently of both the functional composition language and the runtime code deployment methods.

Example

Consider how a typical localization service request is represented in our architecture. To be self-sufficient, the contract includes a reference its execution method, e.g., a compiled library implementation of the localization algorithm, the type of sensors used, such as distance measuring or angle of arrival, and data types for the output (locations and error intervals). To be minimally constrained, it must not specify a deployment location (node ids) or method (a specific range measurement service), referring instead to the contracts in the repository. Execution-specific information is filled in at run-time based on the specified constraints.

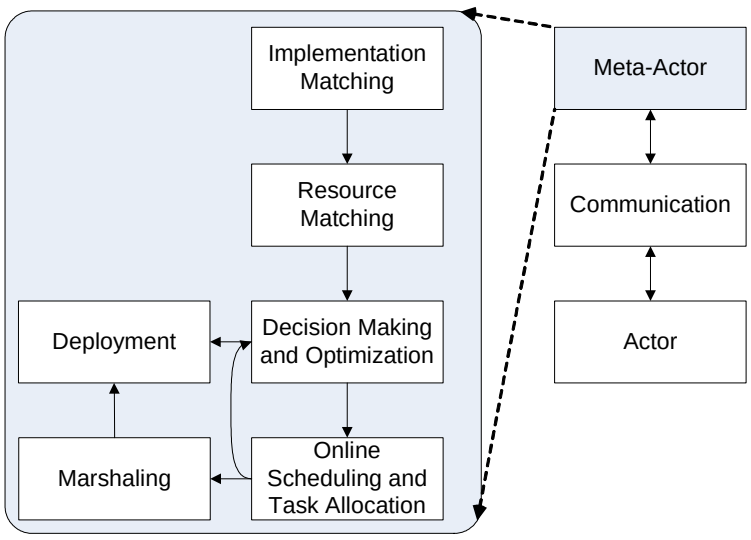


Figure 3.8: Self-mediated execution architecture for middleware services.

3.3.2 Architecture Components

Given an application comprising a composition of service requests represented in such manner, its execution consists of a self-decomposition and self-deployment process. This results in a system of distributed interacting meta-actors responsible for handling the interaction among the services. Execution proceeds concurrently and asynchronously as the preconditions for the deployment of each service request are satisfied. We call this process *self-mediated execution*.

Let us now focus on the role of the meta-actors in this process. Figure 3.8 highlights the governing behavior of a meta-actor in processing service requests. Due to service request self-sufficiency, each meta-actor can decide *how*, *where* and *when* to execute its associated service. We now explain the function of each component of this architecture and their interactions.

Choice of Implementation

Deciding *how* to execute the service request involves matching a particular service implementation to an interface from the service contract repository, and then finding the network resources required by that implementation.

Implementation Matching. This component finds all implementations that match the constraints of a given service request. For example, we might search for implementations of a ranging service with a MeasureDistance method that also satisfy a maximum distance constraint. This is done by querying the contract repository and filtering the results according to the constraints specified in the service request.

Resource Matching. Likewise, the resource matching component finds all suitable resources for a given service implementation. Matching algorithms used by this service depend on the resource description language employed by the system. Several methods are available for indexing a dynamic set of geographically distributed resources, including a yellow pages service, tuple spaces and actor spaces. For instance, if tuple spaces are used, sensor nodes entering the system can publish their resource descriptions in the tuple space, and the resource matching component performs a search in the form of pattern matching [11]. Caching and prefetching techniques can make the process more efficient, eliminating the need to scour the network for each query. Due to the location-dependent nature of most WSN computations, we expect most queries to be limited geographically, avoiding the need to flood the network even in cases when cached information is unavailable.

Location and Deployment

Second, the meta-actor needs to decide *where* to execute the service request. For the sake of efficiency, deployment and invocation are treated separately. As such, code deployment starts as soon as possible, while the invocation is delayed by the scheduling component until the necessary resources become available.

Decision Making and Optimization. Given a list of possible resources and implementations, this component chooses which implementation/resource combination best fits the application requirements or system performance considerations. The output of this service is a platform-specific executable code segment, along with a list of its required resources, which dictate where in the WSN the service must be located. This component comprises the core of the self-mediated execution approach. Choosing an appropriate option from a list of resources and service implementations is critical to efficiently executing composite service-based applications.

Deployment. This component is responsible for transporting the executable code segment to the destination platform, thereby making the service available to other services and applications. If an implementation of the service is already available at the destination platform, the code deployment step is skipped entirely, and the service request is sent to the deployed service.

Scheduling

Third, the meta-actor decides *when* to execute the service request. This is accomplished by the scheduling and task allocation component.

Online Scheduling and Task Allocation. The goal of this component is to decide when the service instance can be deployed and executed. If the resources required by the service instance are not immediately available, its execution is postponed, along with all services that depend on it. Shared resources requiring exclusive access, e.g., certain types of sensors and actuators, must be scheduled globally, since service implementations may not be aware of each other. An up to date resource use schedule is provided to the decision making and optimization component to facilitate the selection of less-utilized resources whenever possible, and a repository of active services is maintained to keep track of all service instances currently deployed in the system. This is also used by the implementation matching component to check if an already-deployed component may satisfy a service request.

Invocation and Execution

Finally, the service request is ready to be deployed and executed on the target platform. This step includes marshaling and remote invocation.

Marshaling. The marshaling component packages the service request for transport and deployment on the destination platform, using the deployment component. The method is platform-dependent. In our system, this involves wrapping the service invocation code in a mobile agent, which can move to the destination node without relying on an external routing service.

The service request is then handed off directly to the run-time environment to launch or query the selected implementation of the service. From this point onward, the service instance interfaces via its actor interface with its meta-actor and with other services in the WSN by means of asynchronous message passing, implemented by the communication component. Asynchronous messaging is used both to deliver computation results and error notifications from the executing services and to deliver control messages from the meta-actor.

3.4 Mobile Code Deployment Platform

Finally, we consider the operational level of the Ambiance platform, where the actual interaction with the cyber-physical system takes place. We employ ActorNet [45], a mobile agent platform for wireless sensor networks, as the mobile code deployment platform. There are several reasons why ActorNet fits well into this role.

Sensor networks are well-suited to multiagent systems: agent autonomy reduces the need for communication, saving precious energy. Mobile agents are also an intuitive technique for remotely reprogramming sensors deployed in the field. However, implementing agent programs directly on a WSN is complicated by the many

limitations of sensor nodes, including limited memory, slow processors, low bandwidth and finite energy. ActorNet eases development by providing an abstract environment for lightweight concurrent object-oriented mobile code on WSNs. As such, it enables a wide range of dynamic applications, including fully customizable queries and aggregation functions, in-network interactive debugging and high-level concurrent programming on the inherently parallel sensor network platform. Moreover, ActorNet cleanly integrates all of these features into a fine-tuned, multi-threaded embedded Scheme interpreter that supports compact, maintainable programs—a significant advantage over primitive stack-based virtual machines.

3.4.1 Actor Language

The top layer of the ActorNet platform is an actor language interpreter. The actor language uses the well-known S-expression syntax. Notable primitive operators include `par`, `send`, `msgq` and `callcc`, which are used extensively in this paper.

The ActorNet language is based on the syntax and semantics of Scheme, with extensions specific to actor computation. The `par`, `send` and `msgq` operators are not found in Scheme. `Par` creates separate actors to evaluate each argument expression in parallel and returns a list of the created actors' identifiers. While these actors remain on the same ActorNet platform, they share parts of their environments, allowing them to communicate efficiently. If an actor migrate to another platform, it can communicate back via message passing. The `send` operator provides a simple, abstract mechanism to send messages to an actor. A deep copy is made of the message data to prevent any dependence on the source host. For example, `(send (list 100 x))` sends all data reachable from variable `x` to an actor with id 100. An actor can access its message queue by calling the `msgq` operator, which returns the list of messages the actor has received.

The `callcc` operator accesses the *current continuation* (CC)—an abstraction of the rest of the program remaining to execute. For example, the CC of the expression `(add 1 (mul 2 ↓ 3))` at the ↓ mark can be regarded as a single-parameter function `c1`: `(lambda (x) (c2 (mul 2 x)))`, where `c2` is an another single-parameter function `(lambda (x) (add 1 x))`. In general, the CC can be regarded as a stack of single-parameter functions. The operand of `callcc` is a single-parameter function: when `callcc` is called the CC is passed to the operand function.

In ActorNet, the CC and the value that will be passed to it form the state of an actor. Because an actor can read its current continuation, it can duplicate itself or migrate to another platform voluntarily by sending its continuation–value pair to another actor. The other actor simply evaluates the continuation on the value to duplicate the sender's behavior. Using these primitives we can easily and intuitively define the agent migration function demonstrated in the next section.

3.4.2 ActorNet Platform

ActorNet is a distributed actor platform targeted primarily at resource-constrained wireless sensor networks. Each ActorNet node is a multi-threaded interpreter for a high-level actor language. An ActorNet network can span several Internet-connected sensor networks, as well as PCs. It consists of three types of nodes: sensor nodes, which directly execute actor code; repeaters, which distribute actor messages in local sensor networks; and forwarders, which bridge ActorNet systems across the Internet and provide an external access point to the actor system. It currently supports the Mica2 and Imote2 sensor platforms, as well as Windows- and Linux-based PCs. Like other virtual machines, the ActorNet platform provides a uniform computing environment for all actors, regardless of hardware or operating system differences; actors can seamlessly migrate between all these hardware platforms.

Actors only use the interpreter module directly; thus implementation details are hidden from actor programs. Lower-level services are necessary to reconcile the desired properties of simplicity and platform-independence in the high-level language with the specifics of the wireless sensor network environment. The layered architecture alleviates some of the complexity of application development for sensor networks, which currently involves a significant amount of low-level programming due to the tight coupling between the application and the operating system. ActorNet aims to provide a stricter decoupling of applications from the operating system, enabling a sensor node to safely load and execute multiple agents, while at the same time simplifying application development.

The ActorNet virtual machine features several services necessary to meet the challenges presented by the WSN environment. These platform services allow efficient memory management and blocking I/O operations for the actor language.

1. *Virtual memory.* Since the 4 KB SRAM space on the Mica2 is insufficient for many applications, ActorNet provides a virtual memory subsystem. It builds a page structure on the serial flash area of the Mica2 and use an inverted page table to access pages stored in an LRU cache in SRAM.

2. *Application-level context switching.* We have devised an application-level context switching mechanism that allows efficient blocking I/O on top of the non-blocking TinyOS model. This mechanism eases development of maintainable applications. To preserve portability and modularity, the context switching mechanism is implemented purely as an application-level service; it does not modify the TinyOS scheduler.

3. *Multi-phase garbage collector.* The ActorNet platform provides a mark-and-sweep *garbage collection* (GC) mechanism. System-level support for garbage collection has many benefits: it eases application

development, eliminates the chance of memory leaks, protects other applications from misbehaving actors, and reduces the actor code size. In order to prevent long-running garbage collection tasks from blocking other applications, we divide the sweep step into many short phases. Combined with the context switching functionality, this greatly reduces the impact of garbage collection on application performance.

3.5 Discussion

We have developed a highly flexible service-oriented architecture for specifying and executing applications represented as dynamic service compositions on wireless sensor networks. This architecture decouples the high-level application design and programming processes from the low-level considerations of the WSN run time environment. This allows the user interface to be presented to the application programmer exclusively in the domain-specific language they are familiar with, while the underlying services and mobile agents can be written in lower-level languages by expert programmers, resulting in highly efficient execution.

Combined with the ActorNet platform for executing the collection of mobile agents representing the dynamic service composition, this architecture provides a foundation for truly dynamic macroprogramming by domain experts. We will explore the development of such a macroprogramming environment in the next chapter.

Chapter 4

Dynamic Macroprogramming

We argued that dynamic macroprogramming by domain experts can be achieved for ambient systems comprising WSNs and traditional computing artifacts, such as PCs and handheld mobile devices, by extending the architectural style of Adaptive Object-Models [91]. The resulting two-level approach to architecting uQuery execution engines allows separating macroprogram representation and reasoning concerns from those of their effective execution on diverse runtime platforms through model-to-code transformation.

We have implemented an end-to-end prototype of the Ambiance dynamic macroprogramming platform. This implementation includes a web-based programmer interface, Dart model for program representation, a uQuery processing engine, and ActorNet-based deployment framework.

A Web-based user interface, built using the Seaside framework [81] in Squeak, has been developed. Programming is done through an XHTML web page, featuring an intuitive drag-and-drop method for selecting and adding services to the application and a spreadsheet-based method for configuring and linking them together (see Figure 4.1). The user interface is fully functional, including support for multiple concurrent users.

A two-level dynamic service composition architecture is implemented as a system of base actors at the operational level and meta-actors at the knowledge level. Using self-mediated execution, meta-actors translate non-executable high-level specifications and instantiate executable base actors, filling in any missing information from the execution context. Representations of queries are transformed to platform-specific code for ActorNet, dynamically deployed and executed.

The presented meta-level and mobile-agent architecture for implementing dynamic macroprogramming is prototyped by reusing and extending our previous work on ActorNet and its implementation on Mica2 motes [45], and Dart implemented as an object-oriented framework in different dialects of Smalltalk [74]. Using ActorNet as the query execution environment provides the dynamicity of macroprogramming, while enabling load balancing and other optimizations to take place. We thereby combine both expressiveness of a Turing-complete language with the simplicity of a domain-related language.

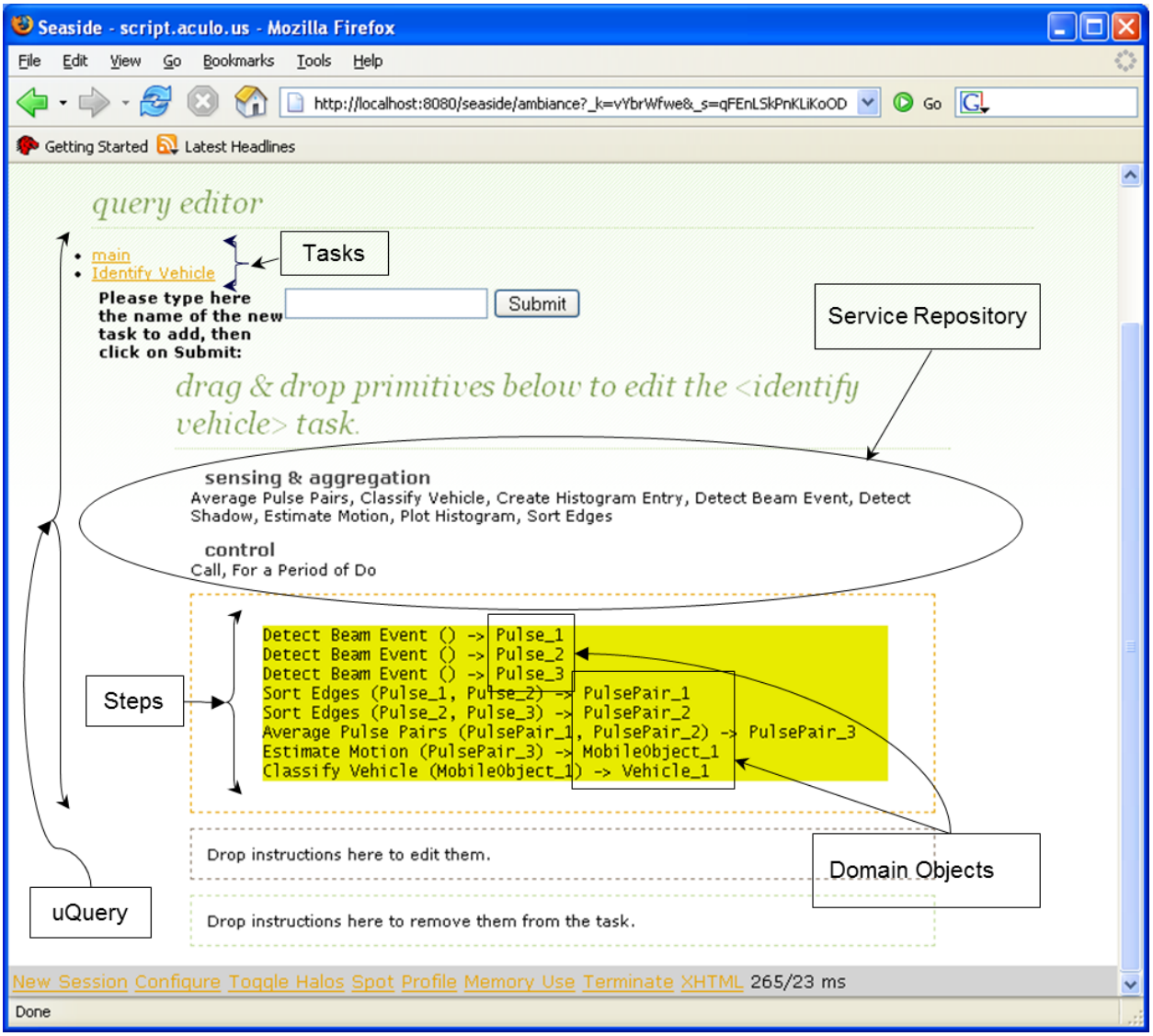


Figure 4.1: Web-based Ambiance user interface.

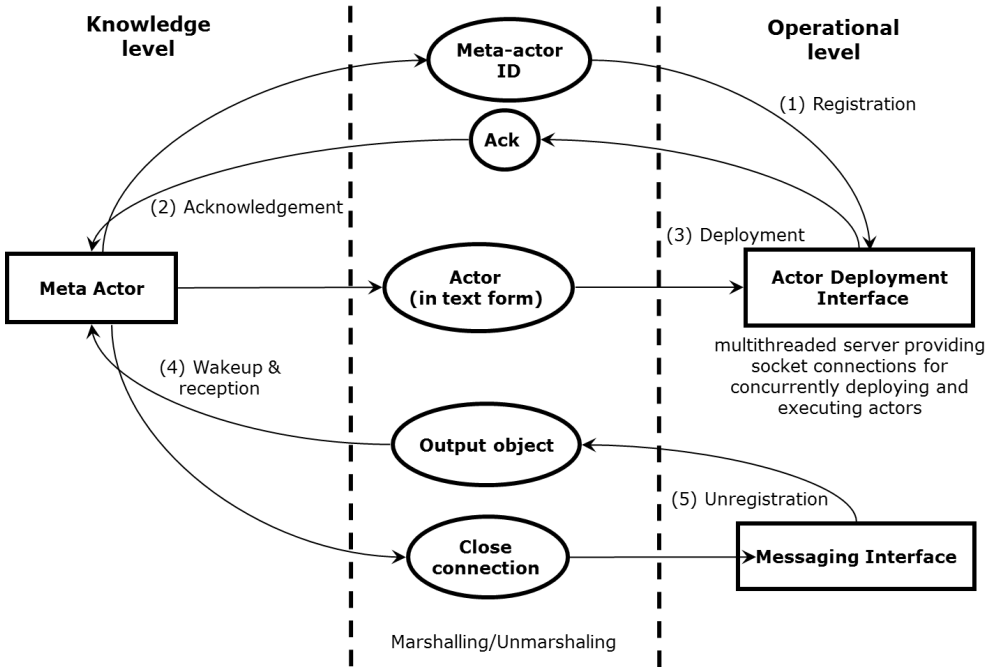


Figure 4.2: Communication protocol between actors and meta-actors.

4.1 ActorNet Platform Extensions

ActorNet was originally developed for stand-alone multiagent WSN applications. Several extensions to the base ActorNet system have been completed to integrate it as the code deployment platform for Ambiance.

The Actor Directory Service (ADS) keeps track of all the application-level actors in the system. It accepts registration messages from actors, and responds to queries from within or outside of the system, returning actor ids (unique actor names). Actors can currently be matched by their output types, but a richer query language may need to be implemented at a later point. The ADS is implemented as a database accessible at forwarder nodes.

An ActorNet forwarder node exports three interfaces to the actor environment, implemented as sockets (Figure 4.2).

- **Actor Deployment.** This interface is used to inject new actor code into the system, instantiate new actors and migrate them to the desired location.
- **Actor Communication.** Sending and receiving messages from the actor system is possible through this interface. Messages can be sent to a specific actor id, or broadcast to all actors in the system.
- **ADS Query.** This interface provides a way for an external entity to look up the deployed actors.

The original ActorNet platform offered only unreliable communication. Since many multiagent applications

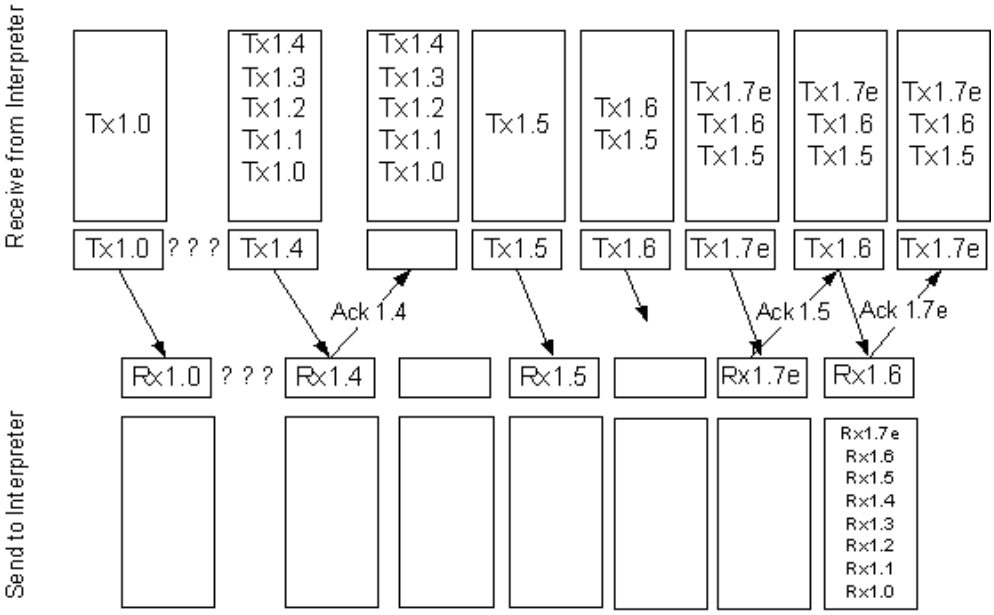


Figure 4.3: Selective-repeat protocol for reliable actor migration.

do not rely on the reliability of a particular actor, this was a good design decision in that context. For actors generated from Ambiance, however, lack of reliable actor migration simply introduces an additional delay, as the system waits for a timeout before launching another instance of the actor.

Two reliable communication methods have been implemented. The first is a selective repeat protocol based on the sliding window concept, similar to TCP. The other is a simpler stop-and-wait protocol sending one packet at a time. We found the stop-and-wait protocol to be superior for environments with low packet loss rates (under 15%), while selective repeat outperformed significantly in lossier environments. Both implementations are now available in the ActorNet platform, and can be selected as appropriate. Figures 4.3 and 4.4 illustrate the behavior of these protocols.

Algorithm 4.1 lists an annotated ActorNet agent. This particular agent had its code dynamically generated by the Ambiance platform. Once deployed on an ActorNet platform through the Actor Deployment Service, its purpose is to migrate to a specific ambient node, execute a primitive step (a low-level service implementation), marshal the output, migrate back to its origin point and deliver the output to its meta-actor via the Actor Communication Service.

4.2 Composable Actors

Composable actors, or services in SOA terminology, are application-level computing entities with well-defined

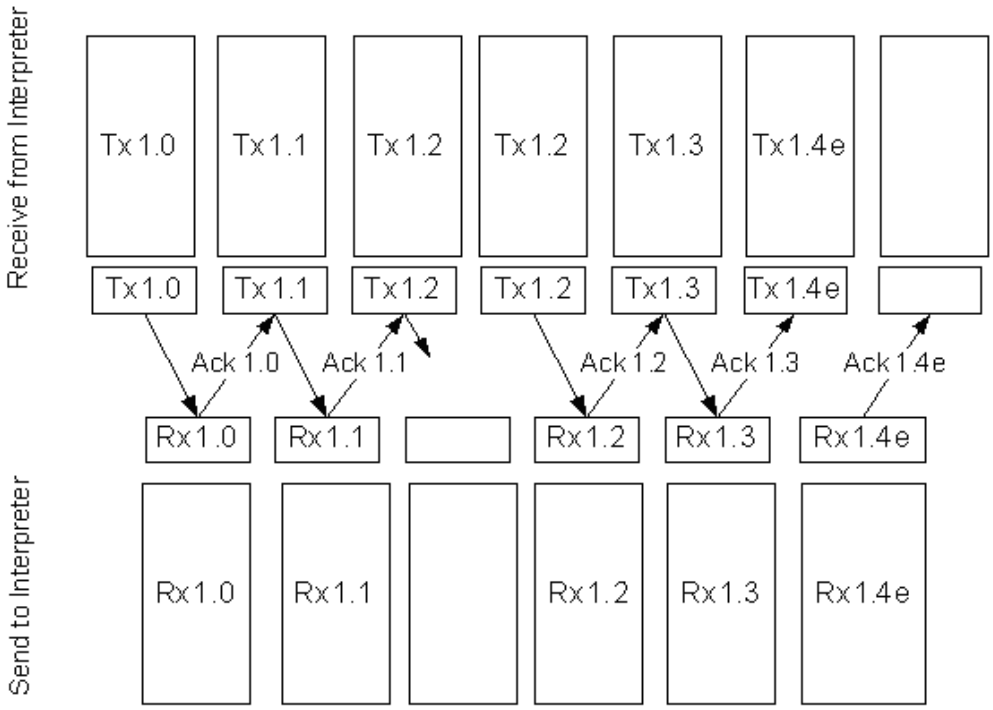


Figure 4.4: Stop-and-wait protocol for reliable actor migration.

Algorithm 4.1 ActorNet agent code generated from a template.

```
 1. (   (lambda (migrate)       ; main function
 2.       (seq
 3.           (migrate          ; migrate to destination
 4.               200           ; destination id
 5.               111)          ; meta-actor id
 6.           (par (extmsg      ; send result back to source
 7.               111           ; meta-actor id
 8.               (migrate
 9.                   100       ; source id
10.                (prim        ; execute primitive
11.                    1        ; primitive index in library
12.                    nil))    ; list of arguments (empty)
13.        ))))
14.    (lambda (adrs val)       ; migrate function
15.        (callcc (lambda (cc) (send (list adrs cc
16.            (list quote val)))))))
```

interfaces specifying the number and types of input arguments, the type of the output, and any constraints applicable to that component. The execution of a macroprogram takes the form of a dynamically constructed composition of such actors. We have implemented a composable actor framework for ActorNet and Ambiance.

The life of a composable actor consists of the following stages.

1. **Registration and Activation.** Separating actor instantiation from the start of actor execution (activation) allows for several possible optimizations. Commonly used actors can be automatically deployed prior to the arrival of a query. Also, the system may use load balancing and other considerations when choosing when and where the actor is to be executed. Once an actor is deployed on an ActorNet node, it registers itself with the Actor Directory Service, and awaits an activation message.

2. **Request Processing.** After an application-level actor is activated, may receive requests for its end result—an object of the specified output type, or a negative result (nil). The actor thus maintains a list of all parties interested in its output.

3. **Argument Collection.** When an actor receives one or more requests, it must produce an output object. In order to do this, it will apply the application logic to the list of arguments it expects. Since some arguments may be produced by other actors, first it is necessary to identify them using the ADS and send them requests for their outputs. When all necessary arguments are thus collected, execution of the application behavior can begin.

4. **Application Behavior.** The application behavior takes the form of applying a Scheme function to the list of arguments, and saving the resulting value. The code for this will be automatically generated based on the specification of the query.

5. **Result Distribution.** Once the result is obtained, it is distributed to all interested parties in the request list. At this point, the actor terminates and may be garbage collected.

Executing a macroprogram on the ActorNet system is then done as follows. First, the program representation is translated by Ambiance into a composition of actors. If any of the necessary actors are not already available in the system, their code is automatically generated from a template. The actors is then be injected into the ActorNet system through the Actor Deployment interface. After the actors are deployed, the ADS needs to be queried for an actor providing the desired result of the query, e.g., a vehicle object. Finally, activation and registration request messages are sent to that actor id through the Actor Communication interface. Once the computation is complete and the result object is produced, it will be sent out as a message through the Actor Communication interface.

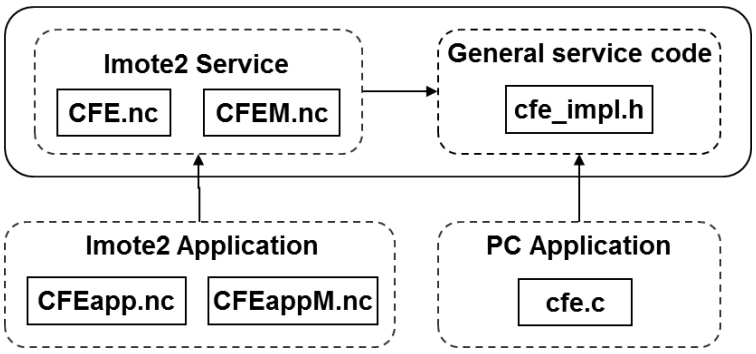


Figure 4.5: PC and Imote2 implementations of the CFE service.

4.3 Application Example

The monolithic SHM application developed by Nagayama and Spencer [62], based on the distributed computing strategy, has been decomposed into constituent self-contained parts. These were transformed into services according to SOA design principles, resulting in a library of modular middleware and application services for SHM. This library has since been expanded and released as an open-source toolsuite for use by civil engineers interested in SHM [36].

This SOA-based approach creates a framework which allows application programmers to more easily create applications for SHM systems deployed on WSNs. As a result, the framework allows more researchers to design and implement successful SHM systems without the requirement of knowing how the underlying middleware and application services are implemented. Both the services provided in the toolsuite and the approach in general are transferable to scenarios beyond SDLV-based SHM.

Services that do not explicitly interact with sensor hardware are designed in a cross-platform manner, such that they can be used either on PCs or ambient nodes without modification. Figure 4.5 illustrates this point for a correlation function estimation (CFE) numerical service. This is an important feature for Ambiance, since it allows the system to choose where to execute these services.

Several key services in this domain have been identified and implemented. *Foundation services* include middleware services used by applications and other services, and are independent of the SHM domain. These include universal sensing, time synchronization, and reliable communication. In addition, *application services*, specific to the SHM problem domain, include algorithms such as correlation function estimation, Eigensystem realization, stochastic damage locating vector (SDLV) damage detection algorithm, and several others [78]. Complete SHM applications can be built from a combination of these services. A more comprehensive description of the SHM problem and a full-scale validation deployment of an SHM system on a 484 m cable-stayed bridge are presented in Chapter 9.

Chapter 5

Mobile Actor Platform

We have introduced ActorNet as the operational-level component of the Ambiance platform, used to distribute mobile actors to specific sensor nodes in the WSN. Let us now delve more deeply into the design of the ActorNet programming language itself [1].

Despite the support offered by middleware services, WSN application development is a complex and challenging endeavor. In this chapter, we identify several issues inherent in programming WSNs under the current state of the art:

- difficulty of adapting applications to new sensor platforms
- lack of support for interoperation between multiple possibly heterogeneous WSNs
- restrictive options for remote reprogramming of sensors
- difficulty of running multiple applications in the WSN

Most embedded systems application developers are familiar with the barriers needed to be overcome to write reliable and efficient programs in this setting. These include challenges in hardware, operating system, and programming languages specialized to particular hardware. To add to the difficulty, new systems are being rapidly developed. Because the users of a WSN may not necessarily be embedded system experts, as Razavi et al. pointed out [75], it is inefficient and costly to hire embedded system programmers to develop and manage WSN applications. A desirable solution might be to provide users a simple and well-known programming language that can hide the complicated details from the users.

Another problem lies in the difficulty of interoperation between multiple WSNs. There are many large scale events that cannot be covered by few patches of sensors. For example, acquiring a temperature map of a city requires a city scale system coverage and often seismic data can be observed across the globe. However, many WSN applications are designed only for a single or a handful of predetermined groups of sensors. Because application images are preloaded on the nodes and the message formats are predetermined, any

[1] Parts of this work appeared in a previous publication [45].

WSNs that are not specifically configured cannot dynamically join the operation. Hence, an application's coverage is bound to a predetermined set of sensors. Also, due to the memory and code size limitations, it is impractical to preload less-frequently used applications on large number of nodes. Another difficulty for interoperation is the heterogeneity of WSN systems. Any WSN that uses a different sensor set, node hardware, or operating system requires a different application image. Thus, adding a new target platform greatly increases the cost and complexity of development and testing.

Without remote reprogramming capabilities, the scalability of WSN services can be an issue. Typically, sensor nodes have very limited memory sizes, and only few services can be compiled together. Because a WSN application image can contain only a fixed maximum number of services, its general utility is limited. Also, this statically programmed WSN is likely to eventually fail to satisfy the dynamically changing needs and requirements of its users. Remote reprogramming capabilities offer one solution. However, unless this service supports fine grained targeting and inter-operation of heterogeneous application images, energy consumption considerations would severely limit its usefulness.

Manual reprogramming of a WSN of any non-trivial size is a laborious and time consuming task. For example, to reprogram a WSN of 100 nodes deployed in a $100 \times 100\,\mathrm{m}^2$ area, one has to walk about one kilometer to collect the nodes, reprogram the 100 nodes, and then walk another kilometer to redeploy them. Furthermore, when the nodes are deployed at an unknown or inaccessible area, manual reprogramming is out of the question. Accordingly, several network reprogramming systems have been developed including Deluge [35], over-the-air programming of Contiki [18], SOS [29], and Trickle protocol of Maté [48]. These systems install the whole image or replace some of the modules remotely injected from a central node. However, some tasks favor a system that not only copies the program but migrates its state to continue its computation on the destination platform. Another concern is that the user may not want to duplicate a program over the entire network or even to copy the whole program. A means to copy only a necessary portion of the code to only a necessary set of nodes is desirable.

One of the design principles of a WSN is to build a large scale distributed system using cheap, even disposable, hardware. Naturally, problems arising from the limited resources follow. For example, Mica2 [31] node has only 4 kB of memory, which is a very tight limit even for a single application. To make matters worse, TinyOS [32], an operating system for the Mica nodes, does not support dynamic loading and unloading of applications. That is, the 4 kB of memory must be shared by *all* applications shipped on a node. These constraints pose a big impediment to the development and the maintenance of WSN applications. Although there are operating systems that support dynamic module/application loading such as Contiki [18] or SOS [29], TinyOS is the dominant operating system for WSNs. According to a recent survey, TinyOS has the largest

support community and the largest number of publications among operating systems for WSNs with 81% [46]. For this reason, our system is implemented primarily targeting TinyOS; however, we believe supporting other platforms may not be difficult given that ActorNet is already providing support for two very diverse platforms: TinyOS on Mica2 and Linux on PC, with only a small fraction of the runtime system code being platform-dependent.

Some of the problems we described above have been studied in the context of *open distributed systems*. The main characteristics of an open distributed system are that adding new components, replacing existing components, and changing the interconnections between components largely do not affect the functioning of the system. Observe that these are desirable features of a computing environment established on WSNs. It has been known that *actor models* are well-established models suitable for the open distributed systems: they have a built-in notion of local component and interfaces, and their asynchronous message based interaction between components prevents the direct control of one component over another, a prerequisite property of open distributed systems. In this chapter, we implement a variation of an actor model, named ActorNet [45], to address some limitations of WSN programming that currently exist. ActorNet is unique among WSN-specific programming languages, mobile agent systems, and virtual machine runtimes due to the options it offers to the application developer for overcoming the challenges described above.

ActorNet is a uniform computing platform for mobile agents, which we call *actors* [3]. ActorNet builds a single virtual network by interconnecting physically separated WSNs over the Internet. This virtual network removes the difficulties in interoperating multiple WSNs together. For example, an actor can track a seismic event while migrating thousands of miles through the Internet if the WSNs are loaded with ActorNet platform. The homogeneity of the computing environments provided by the interpreter layer of ActorNet is another factor that simplifies the interoperation. Because the underlying platform differences and the network differences are hidden from the actors, the same actor program can continue its tasks while migrating between different ActorNet platforms. ActorNet platforms are currently available for Mica2 sensor nodes and for PCs.

The top layer of the ActorNet architecture is an actor language interpreter that insulates actor programs from some of the differences in underlying platforms. The actor language of ActorNet has a simple and well known Scheme-like syntax called *S-expression* [20]. Its prefix notation expressions do not have many special cases, and thus simplify the interpreter implementation as well as lower the barrier of learning the language. Despite its simplicity, the actor language supports powerful operations such as high-order functions, reflection, garbage collection, and tail recursion removal. The specific details of the underlying hardware and the operating system are hidden behind the high level operators of the actor language. The uniform

computing environment also simplifies the application developments greatly as it precludes the need for different variations of programs for different platforms.

The mobility of actors enables a fine-grained network reprogramming. The actors run only at the required nodes and their migration does not disrupt the continuation of other actors' computation. To support actor migration, we uniformly represent the state of an actor as a pair of a continuation [1] and a value to be passed to the continuation. This state representation, along with the reflection capability of the actor language, endows actors with the *voluntary migration* capability. A migration is as simple as building an actor state and sending it to another ActorNet platform. In addition to the actor migration, this uniform actor state representation simplifies the management of concurrent runs of multiple actors in a single ActorNet platform. The choice of the Scheme-like syntax is another simplifying factor for the actor migration. In this syntax an actor program is a list data which can be treated like normal data.

To overcome the limitations of sensor nodes, ActorNet provides a runtime platform with a library services such as virtual memory, application level context switching, garbage collection, and a communication stack machine. These services not only secure the necessary resources for ActorNet applications to run, but also enables ActorNet, as an application running on a sensor node, to coexist harmoniously with other native applications.

5.1 ActorNet Design

The primary goal of ActorNet is to address the problems in WSN application development by providing a uniform virtual computing environment for creating dynamic, light-weight distributed applications. ActorNet is based on a foundation of mobile agents, which provide the necessary features of code mobility, coordination and concurrent operation. In this section, we describe the overall design of the computing environment and highlight the issues that need to be addressed in its implementation.

Let us begin this section by summarizing the principal features that ActorNet should provide to address the problems identified above.

- A light-weight mobile agent platform for WSN systems
- A virtual network platform that encompasses multiple physical WSNs and PC platforms without exposing the hardware and networking differences to the application
- Platform support for multiple concurrent actors, which can freely migrate without interfering with each other

- Programmable in a language that is simple, well-known, and expressive
- Powerful programming tools such as higher-order functions, reflection, tail recursion removal, and garbage collection
- A library of useful services, including a large virtual memory space to dynamically load and run non-trivial actor programs and an application-level context switching mechanism to enable blocking I/O and fair scheduling

5.1.1 Network Architecture

Simplifying the interoperation of multiple physically-distributed WSNs is one of the design goals of ActorNet. Toward this goal, ActorNet builds a single virtual computing environment for mobile actors that encompasses multiple WSNs. Specifically, this environment is constructed by interconnecting the base stations, or gateway nodes, of WSNs via an Internet overlay. Using the virtual environment, differences in the communication network as well as the underlying computing platform can be obscured from application-level actors. Being exposed to these differences, an actor program would have to prepare different sets of handlers for each hardware configuration, which results in duplicated code, unnecessarily complex implementations, and large application code sizes.

The virtual environment spans two tiers of networks: an *ad hoc* wireless network and the Internet, as can be seen in Figure 5.1: the *forwarder* turns the Internet into a single-hop, broadcast overlay network, and the *repeaters* act as a bridge between it and the ad hoc wireless networks. This network architecture obscures the underlying network differences from the actor programs. These two network tiers feature vastly different topology, bandwidth, protocols, and performance characteristics. In *ad hoc* wireless networks, messages are locally broadcast to a node's neighbors, whereas most of the Internet consists of wired, point-to-point connections. The bandwidth differences between the two network types can be huge. Typical RF network devices used to interconnect wireless sensors can communicate at speeds ranging from 38.4 to 250 kbps, for 802.15.4 devices. However, in practice the communication speeds are much lower. For example, a 64-node WSN deployed on the Golden Gate Bridge took 12 hours to transport 90 seconds worth of high-frequency vibration data [42]. Finally, there is a multiple order of magnitude difference in performance of the hosts comprising the network: personal computers (PCs)and servers connected to the Internet typically have processors running at several GHz, whereas sensor nodes feature processors with maximum speeds of several MHz. These differences make developing applications that span both network types a challenging endeavor.

To hide the heterogeneity of the network from the actors, we represent the Internet a single hop broadcast

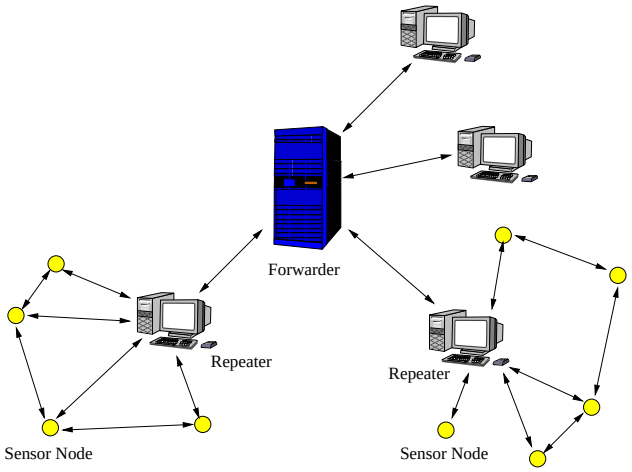


Figure 5.1: ActorNet network architecture.

network from the actor's viewpoint. Specifically, any messages transmitted from a *gateway node* connected to the Internet are forwarded to the other gateway nodes. In view of this network topology, the difference between the *ad hoc* network and the Internet is hidden from the actors. All ActorNet platforms connected to a forwarder node over the Internet can be regarded as a single hop neighbor. This virtual single-hop network extends the range of mobile actors to the global scale. That means, to an actor, a migration of thousands of miles through the Internet is no different from a local migration between two neighboring sensor nodes. As a solution for the network bandwidth differences, ActorNet provides a packet buffer at the gateway nodes, which compensates for temporary differences in the throughput of the two networks.

5.1.2 Actor Language

Like most embedded systems, developing applications for WSNs has a high initial barrier of learning the new system, including the limitations of the hardware and the details of the hardware-tailored application development environments. We design the actor language to lower this initial barrier by hiding most hardware-specific details from the programmer. For example, a simple `send` operator can be used to send any type or volume of data to any destination node, even if it resides in a different WSN connected via an Internet gateway. The language itself has a Scheme-like simple syntax, which has been recognized as among the easiest to learn and has powerful abstraction capabilities.

Another concern in the design of ActorNet is to provide a uniform computing environment to the actors in the presence of the heterogeneity in the computing platforms. The actor language interpreter of ActorNet naturally shields the platform differences from actors as it functions as a virtual machine. However, the interpreter does not automatically guarantee code mobility across heterogeneous platforms; the actor state

should also be maintained across the platforms. We explicitly manage the state of an actor as a pair of a *continuation*, which is a single parameter function that represents the rest of the program [1], and a *value* to be passed to it. Applying the value to the continuation produces a new actor state, and by repeatedly generating new states, an actor makes its computation. Because an actor program can be represented as a list data type in our actor language, this actor state representation is compatible with different underlying platforms. This explicit state representation also helps in simplifying the implementation of concurrent execution of multiple actors on a single ActorNet node.

Voluntary actor migration must also be taken into account in the design of the actor language. Again, the explicit representation of actor states plays a crucial role in actor migration. With the continuation-value pair representation, an actor migration is as simple as sending an actor state to a new platform and applying the value to the continuation at the destination. The reflection feature enables actors to migrate by themselves at any state of their computation; through the reflection, an actor can access its continuation, and at the same time the actor knows the value to be passed to the continuation. Thus, an actor can obtain its state and send it to a new platform. To make the migration happen, ActorNet platforms need to be ready to receive the actor states and let them continue their computations. For this purpose, each ActorNet platform features a special-purpose built-in actor that receives such messages and creates a new actor to evaluate the message content. During the evaluation, the new actor's state is soon be replaced with the actor state in the message.

5.1.3 Runtime Platform

Running ActorNet platforms on sensor nodes presents its own unique difficulties. In this section, we describe the concerns in developing ActorNet platforms on extremely resource-constrained sensor platforms, such as Crossbow Mica2 sensor nodes. The same feature set would be much easier to implement on more powerful sensor platforms, this approach provides the broadest coverage of commonly used WSN devices. As background for this discussion, we briefly present the features the Mica2 hardware and its operating system, TinyOS.

Mica2 and TinyOS

The Crossbow Mica2 mote is built on an 8 MHz 8-bit ATmega 128L CPU with 4 kB of SRAM, 128 kB of program flash memory, and 512 kB of serial flash [31]. The 4 kB SRAM space is shared by the stack, heap, and static variables of all TinyOS components and applications. This, in turn, places a tight memory constraint on applications. Application code, large constant tables, and log data are loaded in the flash memory units. As an application, ActorNet also has to share this 4 kB space, but because its data is actor programs, the small memory is the more restrictive to ActorNet compared to other applications. To address

this fundamental problem, we designed a 56 kB virtual memory formed at the 512 kB serial flash memory. Usually, flash memory read operations are fast, but write operations are slow and expensive in terms of energy consumption. On Mica2 it takes ∼15 ms to write a 128-byte page to flash.

TinyOS is a lightweight operating system for the sensor nodes written primarily in NesC [25]. The system is structured as a collection of modules which are statically linked together based on a component specification. The modules consist of statically-allocated variables and three different types of program blocks: `command`, `event`, and `task`. Service requests are typically split-phase: a caller invokes a command, which returns quickly; once the request is satisfied, the service calls back to a corresponding event procedure in the caller. This communication pattern enables a higher application throughput as compared to simple blocking I/O. Long-running procedures are explicitly executed as *tasks*, which are scheduled in series and run to completion. Since only interrupts can preempt tasks or lower-priority interrupt handlers, if multiple processes must be run concurrently, they have to be explicitly segmented into a sequence of tasks.

In order to enable the mobility of actors, ActorNet supports dynamic loading and unloading of actor programs. Because an actor may allocate and deallocate memory during its computation, the dynamic unloading module must reclaim all the dangling memory made by the unloaded actor. As a general solution to this problem, we added a *Garbage Collection* (GC) mechanism to ActorNet. The GC mechanism is based on the mark and sweep GC algorithm, which is effective but induces an unpredictable latency with a large mean and a large variance. While the GC is running, most of the services are stopped, which can be critical for periodic sampling or communication services. The large variance in the latency also prevents an efficient task scheduling. As a remedy to these limitations, we developed a multi-phase GC algorithm. Assuming that the virtual memory is lightly loaded and thus the mark phase is fast, we divided the sweep phase into multiple sub-phases and deallocated only fractions of the memory on each step. This partial deallocation reduces the slow flash memory write while maintaining the page hit ratio high. The multiple phase GC algorithm also reduces the mean and the variances of the GC latency, which results in a better scheduling.

TinyOS achieves concurrency among applications through split-phase programming. More specifically, most of the I/O operations are supported only in the split-phase style. Although this style of programming increases throughput, the limited support of the blocking I/O makes it difficult to develop and maintain applications. For example, let us consider the code in Figure 5.2. The code on the left side is written with blocking I/O: `bar` calls `foo` and `foo` calls `read` which performs an I/O. Without blocking I/O, we must split the functions as in the right side of Figure 5.2: an application calls `prebar` and arranges an *I/O completed* event handler to call `postbar`. The problem is that every possible function call chain reachable to `read` should be divided into two parts like Figure 5.2. Furthermore one cannot use *stack allocated local variables*

```
foo() {
    int a;
    ...
    read();
    ...
}
bar() {
    int a;
    ...
    foo();
    ...
}
```

```
int foo_a;
int bar_a;
prefoo() {
    ...
    preread();
}
postfoo() {
    postread();
    ...
}
```

```
prebar() {
    ...
    prefoo();
}
postbar() {
    postfoo();
    ...
}
```

Figure 5.2: Code examples with (left) and without (right) blocking I/O.

across the divided functions. That is, all such variables must be declared as static variables which take up space even after the functions are returned. The problem is even graver in ActorNet: any memory access can make a page fault which leads to a flash memory access, an I/O operation; split phasing on every memory access is practically impossible. To overcome these problems, we implement an application level context switching mechanism. It enables ActorNet to return control to TinyOS and regain control later with the same register, flags, and stack configurations. Using this mechanism, ActorNet enables blocking I/O for its actors. This mechanism also provides a seamless concurrent execution of ActorNet alongside other applications. The context switching mechanism is developed at the application layer to reduce the difficulty of porting it to other platforms.

The state representation of an actor can be structurally complicated. Specifically, there can be loops or multiple references in the list structure. Thus, sending and receiving the list data involves serialization and deserialization. One of the concerns in sending a message is that the message is locally broadcast to all the neighbors of the sender. Having a single sender and multiple receivers, it would be computationally beneficial to put as much complexity to the sender as possible and make the receiver simple. In order to utilize the sender/receiver imbalance, we designed a simple communication stack machine: a sender handles all the complexities of communication and makes a stream of data mixed with stack manipulation commands such that the receivers can restore the data by simply running their stack machine by following the commands.

5.2 Example

Before going into the details of the actor language and the platform implementation, let us briefly examine a complete example application to better understand the design of the ActorNet platform.

Consider an actor migrating through a WSN in search of a local temperature maximum—a typical

environmental monitoring task. The actor in this example autonomously selects its migration path based on the environmental information and reports the final result to a base station. The example demonstrates the high level of abstraction for WSN application development provided by ActorNet. An outline of the application is as follows.

1. An actor A broadcasts to its neighbors a small actor that measures the temperature at a node and sends back the result.

2. A determines the local maximum temperature and migrates itself to the corresponding node. When it migrates to another node, A records its point of origin so that it can forward the maximum temperature reading back along the path it followed.

3. When it arrives at a point of maximal temperature, A migrates back to the base station. Upon arrival at the base station it prints out the temperature value.

Note that, in this example, we do not need any platform or OS support for multi-hop message routing. An actor locally broadcasts and moves itself to its neighbor with the greatest temperature, while constructing the return path as it migrates from node to node.

Let us first consider a `migrate` function that makes an actor migrate to another node and then continue its execution. Recall that the state of an actor is a pair of a continuation and a value to be passed to the continuation. In ActorNet, an actor can easily migrate itself to a neighboring node, using the explicit state representation and sending its *current continuation*. There is a `launcher` actor running on every ActorNet platform that receives the messages sent to it as programs and evaluates them. The entire actor migration process is implemented using this very short `migrate` function.

```
1 (lambda (address value) ;; migrate
2   (callcc
3     (lambda (cc)
4       (send (list address cc (list quote value))))))
```

The code for the temperature-search example, which utilizes this migrate function, is listed in Algorithm 5.1. The precise syntax and semantics of the language will be described in the next section, but this code excerpt illustrates the general structure of an ActorNet program and the compactness of the actor language. Note that a relatively complex application is implemented in under 20 lines of code.

The program first broadcasts a `measure` actor that reads a temperature at a remote node and sends back the reading. The sender then waits for 10 seconds and then checks its message queue, `msgq`, for the measurement. No other work is needed for synchronization. The `measure` actor can be encoded simply as

Algorithm 5.1 Actor program to locate and return maximal temperature in a WSN.

```
 1 (rec (move path temp) ;;return path, max temp
 2  (seq
 3    (send (list 0 measure (id))) ;;broadcast measure actors to neighbors
 4    (delay 100) ;;wait for 10 seconds
 5    ( (lambda (maxt)
 6        (par
 7          (cond (le (car maxt) temp)   ;;if it arrives at a maximal point
 8            (return migrate path temp) ;;then return the temp along the path
 9            (move                      ;;else move to the highest temp.  node
10              (cond (equal path nil)
11                (cons launch path)
12                (cons (io 0) path))
13              (migrate
14                (cadr maxt)   ;;node id
15                (car maxt)))) ;;temp
16          (setcdr (msgq) nil))) ;;reset msgq
17      (max (cdr (msgq)) (list 0 0))))) ;;find the max temp.  and the node
```

```
1   (lambda (ret) ;;measure
2     (send (list ret (io 1) (io 0)))).
```

The `(io 1)` system call returns a temperature reading and the call `(io 0)` returns the unique node identifier. A `launcher` actor running at a remote platform will evaluate this function with the return address, which is the `id` function call of the 3rd line of Algorithm 5.1. Although the `measure` actor in this example is simple, it could be an arbitrarily complex function. That is, an actor can easily distribute a complex piece of its code to run in other nodes and later collect the results in the form of messages. This example demonstrates the versatility of ActorNet as a concurrent computing environment for multiple actors.

Returning to Algorithm 5.1, the `move` function takes a return path and the current maximum temperature reading as its parameters. Migration occurs after evaluating the second parameter. Line 9 shows how the actor migrates to another node: it first appends its node id—`(io 0)`—to the return path and then migrates to the node where the greatest temperature was read. When the actor arrives at a point of the maximal temperature, it returns the temperature value using the `return` function, listed below.

```
1 (rec (return migrate path temp)
2   (cond (equal path nil)
3     (print temp)
4     (return migrate (cdr path)
5             (migrate (car path) temp))))
```

The `return` function is similar to `move`. It migrates across the nodes along the return path.

Note how easy it is to write a mobile agent program using the ActorNet platform. By providing simple-to-use and high-level features, ActorNet enables a rapid development of powerful WSN applications. Furthermore, because mobile agents operate autonomously, they can be used in resource-constrained sensor networks that do not provide many supporting services.

Finally, it is worthwhile to mention that this program does not require any routing services: the actor follows a steepest temperature ascent path, and it also maintains a return path by itself. Also note that the application does not require collecting the temperature reading from all nodes to a central node (usually done by a data dissemination process); instead the actor collects and processes the information while migrating in a sensor field. Even considering the data aggregation service, the saving in the amount of communication by the mobile agent approach is very significant.

5.3 Actor Language

In this section, we describe the syntax and semantics of the ActorNet actor language. ActorNet expressions have the well-known *S-expression* syntax defined as follows.

$$
\begin{aligned}
\mathcal{E} \quad ::= \quad & \mathcal{N} \mid \mathcal{S} \\
\mid \quad & (\texttt{lambda}\ (\mathcal{S}^*)\ \mathcal{E}_{body}) \\
\mid \quad & (\texttt{rec}\ (\mathcal{S}\ \mathcal{S}^*)\ \mathcal{E}_{body}) \\
\mid \quad & (\texttt{cond}\ \mathcal{E}_{test}\ \mathcal{E}_{true}\ \mathcal{E}_{false}) \\
\mid \quad & (\texttt{quote}\ \mathcal{V}) \\
\mid \quad & (\mathcal{E}_{op}\ \mathcal{E}^*),
\end{aligned}
$$

where $\mathcal{E}_{body}, \mathcal{E}_{test}, \mathcal{E}_{true}, \mathcal{E}_{false}, \mathcal{E}_{op}$ are expressions. When a value term is structured as above, it is given with a sort $\mathcal{E}$.

Like Scheme, our actor language uses prefix notation. For example `(add 1 2 3)` returns 6. Actor language has arithmetic operators like `add, sub, mul, div`, and logical operators like `and, or, not`. It also has a set of pair and list manipulation operators. For example `(cons 1 2)` returns a pair of 1 and 2, and `(car (cons 1 2))` returns the first element 1, and `(cdr (cons 1 2))` returns the second element 2. `(list 1 2 3)` returns a list $(1, 2, 3)$ which is equivalent to `(cons 1 (cons 2 (cons 3 nil)))`. Note that `(cdr (list 1 2 3))` is $(2, 3)$. There are assignment operators `setcar` and `setcdr` that set the first and the second elements of a pair.

An expression beginning with `lambda` is an anonymous function definition, where $\mathcal{S}^*$ are zero or more names for the function parameters. To ease writing recursive functions, the actor language has the `rec` primitive, where $\mathcal{S}$ is for the name of the function and $\mathcal{S}^*$ is for the parameters. `cond` is used for branching expression: if $\mathcal{E}_{test}$ is evaluated to be true, $\mathcal{E}_{true}$ is evaluated; otherwise $\mathcal{E}_{false}$ is evaluated. Observe that this behavior is not the call by value semantics of the function application. `quote` is an operator that returns its parameter as a value without evaluating it. This operator is useful when we are sending a list as a data to another ActorNet platform. Without this operator, building a literal list is difficult because the interpreter regards the literal list as a function application and tries to evaluate all the elements. The `seq` operator is similar to the `begin` operator of Scheme: each expression is evaluated in turn, and the value of the last expression is returned.

The `par, send` and `msgq` are new operators not in Scheme. `par` creates new actors for each expression and makes the actors evaluate the expressions in parallel. The return value of the `par` expression is a list of the created actor ids. While these actors remain in the same ActorNet platform, they share some parts of their environments so that they can communicate efficiently. If actors migrate to another platform, they can communicate via asynchronous messages. The `send` operator provides a simple mechanism to send messages to an actor. `send` makes a deep copy of the message and transmits it to the receiver to prevent any dependence on the source host. For example, `(send (list 100 x))` sends all the data reachable from the variable `x` to an actor with id 100. An actor can access its message queue by calling the `msgq` operator, which returns the list of the messages the actor has received. ActorNet internally uses a `recv` method that receives the massage and collects it to the list returned by `msgq`. Note that `msgq` is one of the operators that makes our Actor language non-functional; it may return different values for different calls.

The `callcc` operator accesses the *current continuation* (CC)–an abstraction of the rest of the program remaining to execute [41]. For example, the CC of the expression `(add 1 (mul 2 ↓ 3))` at the ↓ mark can be regarded as a single-parameter function `c1: (lambda (x) (c2 (mul 2 x)))`, where `c2` is an another single-parameter function `(lambda (x) (add 1 x))`. In general, the CC can be regarded as a stack of single-parameter functions. The operand of `callcc` is a single-parameter function to which the CC is passed.

In ActorNet, the state of an actor is a pair of a CC and a value to be passed to it. Because an actor can read its current continuation, it can duplicate itself or migrate to another platform voluntarily by sending its continuation-value pair to another ActorNet platform. By simply applying the continuation to the value, the sender's computation is continued on a new platform. Using these primitives we could easily and intuitively define the `migrate` function of Section 5.2.

5.4 ActorNet Runtime Implementation

Based on the design presented in Section 5.1 and the language definition of the previous section, we discuss the issues in implementing the ActorNet runtime platform.

5.4.1 Network

ActorNet provides a single virtual WSN to actors by connecting physically separated multiple WSNs through the Internet. Recall that the uniform network structure of the virtual WSN is ensured by making the Internet a single hop broadcast network. ActorNet implements two services called *repeater* and *forwarder* to build the uniform network. The repeater bridges the communications between the Ad-Hoc wireless network and the Internet by passing all messages received from one network to the other. Meanwhile, the forwarder provides a single-hop broadcast overlay over the Internet by replicating the messages from each repeater to each of the others connected to it. The net effect of the repeater/forwarder architecture is transforming the individual physically-separated WSNs into a single-hop neighborhood.

Figure 5.1 shows the repeater/forwarder network architecture of ActorNet. Each repeater has a node called GenericBase through which the repeater can hear from and talk to its WSN. A repeater injects any message it hears from the Internet into its WSN and it sends any message it overhears from its GenericBase to the forwarder. On the other hand, a forwarder is listening to a TCP port for any connections. Once a connection is made, the client is registered to the forwarder until the connection is terminated. In summary, any message overheard by a repeater from its WSN is transmitted to the forwarder and then retransmitted to the other repeaters and ActorNet platforms running on PCs. Finally, the messages sent to the repeaters are injected into their WSNs.

The network bandwidth difference problem is currently handled by placing a large message buffer at the repeaters. The fast messages from the Internet are gathered at the buffer and then slowly retransmitted to the WSNs. However, as the number of clients to the forwarder is increased, the repeaters will constantly send messages to their WSNs. This will increase the chance of a network collision and drain the energy from the nodes near the GenericBase. In addition, the buffering solution is only valid while the input data rate to the buffer is smaller than its output rate. To address these problems, a smarter scheme that makes the repeaters selectively filter the messages can be used. The filtering is based on the actor computation model: when an actor is created, its unique id is known only to its creator, and as the parent or the children send messages with the new actor ids, others can communicate with the newly created actor. Thus, unless any messages with the actor id have passed through a repeater, no actors at the other side of the repeater know the existence of the new actor. Because every data is associated with its type in actor language, the actor id

checking at the repeater can be effectively done by adding a new type for the actor ids. Observe that the actor ids stored in repeaters can be regarded as the receptionist names and the external actor names of the *actor configuration*.

5.4.2 Language

Recall from section 5.1 that an actor state is a pair of a continuation and a value, and the computation of an actor is a series of actor state transitions made by applying a state's value to its continuation. In ActorNet, these state transitions are implemented by two core methods of actor language interpreter called `eval` and `apply`. `apply` takes the continuation and the value of an actor state as its parameter and produces a new actor state by applying the value to the continuation. `eval` takes an expression and an environment as its parameter and evaluates the expression within the environment. While evaluating an expression, the values of the identifiers are looked up from the environment that actually is a stack of identifier-value pairs. The environment stack is stored at a structure called *closure* when `eval` encounters a function definition, and is extended when the actual function parameters are applied to the function. The stack structure of environment and the use of closure ensure the lexical scoping rule when looking up the identifiers.

The tail recursion is a recursive call that does not necessarily build up the states [1]. Because actor programs use recursions for the loops, the tail recursion removal is crucial for actor programs; without it, the stack will grow for any simple loop implementations. ActorNet implemented only the basic tail recursion removal capability: the return addresses of function calls are eliminated. Although it is not a fully optimized capability, loops can be effectively replaced with parameterless functions. The return addresses are naturally eliminated through the use of the continuation in the actor computation.

The computation of an actor is explicitly managed as transitions of actor states. This explicit state management leads to a simple and notationally clean implementation of multi-threading capability. Because all the necessary information required to proceed the computation of an actor is stored in the actor state, the context switching is as simple as taking an actor state from a queue of actor states and then applying its value to its continuation. This mechanism is similar to the trampolining technique [1], except that ActorNet schedules the switching and the states are explicitly managed. Observe that the environments play the role of the stack, but they are essentially linked lists, as oppose to linear arrays, built on the virtual memory. This dynamic structure eliminates the stack management during the context switch.

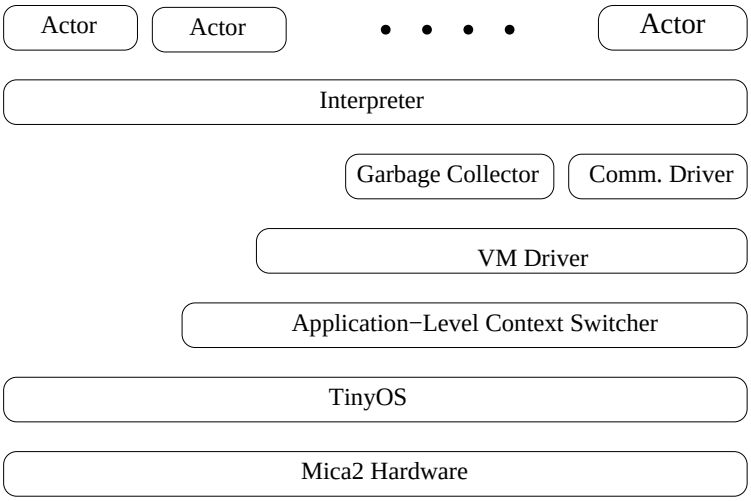


Figure 5.3: Software architecture of ActorNet platform on the Mica2 sensor node.

5.4.3 Platform

The current implementation of the ActorNet platform is implemented in only 30 kB of code and 2 kB of data. The code is stored in the Mica2's 128 kB flash memory unit, leaving 100 kB for other applications; the data is allocated in the 4 kB of SRAM space.

Figure 5.3 depicts a layered software architecture of ActorNet platform for a sensor node. A module does not know the modules above it, but it has access to all the modules below it, not just the ones immediately below it. In contrast, actors only use the interpreter module. Thus, the implementation details are hidden from the actor programs.

Virtual Memory

ActorNet provides a *virtual memory* (VM) subsystem which uses 64 kB of the 512 kB serial flash as the virtual memory. This address space is efficiently indexed by a 16-bit integer. The virtual address space is divided into 512 pages of 128 bytes each. In addition, 8 pages of SRAM (1 kB) are used as a cache for the virtual memory. While flash is not commonly used as a virtual memory store due to the limitation on the maximum number of writes it supports (typically about a million writes to each location), the relatively slow operating speed of sensor nodes and small data sizes of mobile actors mean that even long-term deployments of wireless sensors are very unlikely to approach this limit.

An inverted page table is used to search the cached pages for a requested address. It is implemented as a priority queue that maintains the 8 most recently used pages. Hence, the page replacement follows the *Least Recently Used* (LRU) policy. Figure 5.4 shows the structure of a page. Including bitmaps in the page

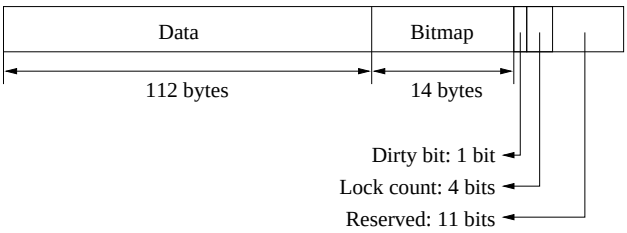


Figure 5.4: ActorNet page structure.

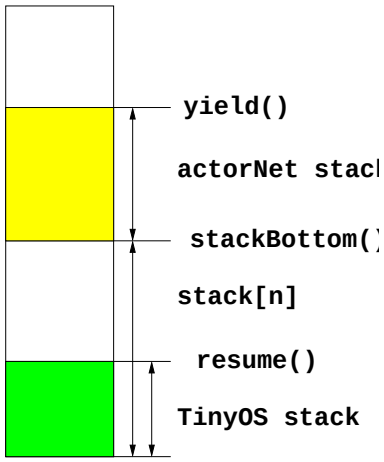
structure imposes a performance penalty when searching for a free space. However, having a smaller RAM space than the size of the bitmap, it is an inevitable decision. The 128 byte page is divided into a 112-byte data area, a 14-byte bitmap, a 1-bit dirty bit flag, a 4-bit lock count, and 11 bits of reserved space. Because the flash memory writes are slow, we used the dirty bit to avoid an unnecessary page writing. The lock count is used to prevent the VM subsystem from swapping out certain pages. For example, the communication driver of Figure 5.3 uses a set of static variables defined in a structure called `ComData`. Because this data has buffers shared with the TinyOS communication subsystem, its container page must be locked during transmit and receive operations. This is accomplished by calling the VM's `lock` procedure, subsequently followed by an `unlock` call.

Since there are 112 bytes of data area per page, the effective virtual memory space is 56 kB (512×112). In Figure 5.4, an allocation bitmap with 8-byte granularity is maintained at the end of each page. Note that the whole 4 kB SRAM space of the Mica2 is not large enough to hold the bitmap of all 56 kB of virtual memory space: $56\,\text{kB}/8 = 7\,\text{kB}$. Distributing the bitmap at each page has a disadvantage when searching for a free space, because the VM driver has to load each page from the flash to check the free space. On the other hand, it is crucial to save the precious SRAM space.

Evaluation based on the benchmark of recursively computing the n^{th} Fibonacci numbers showed a page hit ratio of 95.00%. However, the page hit ratio rises to 99.06% if we consider only the page misses involving the flash-write operations.

Application-Level Context Switching

Figure 5.5 shows pseudo-codes of `yield` and `resume` methods for the context switching mechanism. In order to perform the context switching correctly, stack contents and register values must be preserved. We reserved a stack space for TinyOS and other applications by defining the `stack[n]` array in the `stackBottom` function. Register values including the *program counter* and *stack pointer* are stored and reloaded through the `setjmp` and `longjmp` system calls. The control flow for this mechanism is as follows.




```
jmp_buf toTos, toApp;
void yield() {
  if(setjmp(toApp)==0)
    longjmp(toTos,1);
}
int resume() {
  int r=setjmp(toTos);
  if(r==0)
    if(/*first time called*/)
      stackBottom(500);
    else
      longjmp(toApp,1);
  else
    return r!=2;
}
void stackBottom(int n) {
  char stack[n];
  /*start ActorNet platform*/
  longjmp(toTos,2);
}
```

Figure 5.5: Application level context switching mechanism.

1. When `resume` is called from TinyOS, it stores its register values in `toTos`. If this is the first time that `resume` has been called, `stackBottom` is called to allocate TinyOS stack space by defining `stack[n]` array. Following stack reservation, `stackBottom` initiates the ActorNet platform.

2. When ActorNet calls `yield`, the current register values are stored at the `toApp` variable and the control flow is returned from the `setjmp` call of the `resume` function. Note that control does *not* go back to the `stackBottom` function: the value of `r` in `resume` is 1 in this case.

3. When the `resume` function is called again from TinyOS, the register values are restored from the `toApp` variable and control flow is returned to the `setjmp` of the `yield` function.

The left side of Figure 5.5 shows the stack configuration with this mechanism. In the figure, the stack fills up from the bottom. The shaded area below the `resume()` is the stack space used by TinyOS. The white area below the `stackBottom()` is the additional stack space allocated to TinyOS in the `stack[`n`]` local variable. We use $n = 500$ for Mica2 platforms and $n = 5000$ for PC platforms. Note that the TinyOS stack is limited to this white area; while, in general, we cannot anticipate a stack usage, the applications running on a Mica2 are fixed when a binary image is loaded. This, combined with the fact that most TinyOS applications do not employ recursion, means that in most cases the stack usage is predictable. The shaded area above the `stackBottom()` is the stack space used by the ActorNet platform. The `yield()` line shows the top of the application stack when the `yield` is called.

```
read() {
  ...
  while(!isFlashReadDone)
    yield();
  return flashData;
}
```

```
task loop() {
  resume();
  post loop();
}
```

Figure 5.6: NesC program for the `read` function.

In order to explore the utility of the context switching mechanism, let us consider the NesC program for the `read` function (Figure 5.6). Note that there is a spin-loop in the `read` function waiting for the `isFlashReadDone` variable to become true.

With our context switching mechanism the `yield()` call in the `read` function causes control to exit from the `resume()` call of the `loop` task. Thus, TinyOS can schedule other tasks and process pending events. Later, when the `loop` task is scheduled again and the `resume` function is called, the computation continues from the `yield()` call of the `read` function as if it had just returned from the `yield`. Note that we do not need to divide the application program into two phases as in Figure 5.2. Hence the yield-resume mechanism improves the maintainability of applications.

Multi-Phase Garbage Collector

We implemented a *mark and sweep* garbage collector to reduce programming errors and to relieve the developers of the burden of manual memory management. Our actor language is a typed language: every data value is tagged with a byte for its type. Be cause there are only a handful of data types in the actor language, the rest of the bits can be used as marking flags. In fact, the garbage collector uses two bits for its marking and the communication stack machine uses another bit for serialization. Marking the reachability of a memory cell from any active actor states can be done easily because all actor states are explicitly managed. However, there are also temporary data produced by the actor language interpreter that are not yet bound to their state. To prevent them from being swept away, ActorNet manages a list of the temporary data until their actor state is updated.

In our experiments, the conventional mark and sweep GC can take as long as 10 seconds on Mica2 nodes. This delay can slow down the communication speed considerably, as flash write operations prevent any other computations, including radio communication, in TinyOS. Due to the memory limit, we cannot allocate enough communication buffers to cover the full 10 seconds of GC. We could squeeze the memory to make a communication buffer for 4 packets, but with the conventional GC algorithm this buffer can allow only 1 packet per 2.5 seconds. Instead, we redesign the GC algorithm to have a shorter latency.

To solve this problem, we divide the sweep step into several subphases. Each subphase clears 10 pages,

which takes approximately 150 ms. If we disregard the mark phase, ideally, we can send as many as 26 packets per second, because ActorNet has a communication buffer for 4 packets. With the multi-phase GC algorithm, there is a transient time that the mark phase is finished, but the sweep phase is not completed for all pages. The memory allocated during the transient time needs a special care. Suppose that we do not mark the freshly allocated memory, then the memories allocated at not-yet-swept pages will be erroneously deallocated later. On the other hand, if we mark the fresh memory, the memories allocated at the already swept page will not be cleared in the next round of GC. To solve this problem, we implement a 2 bit marking scheme. In this scheme, we alternate the marking bit on each GC round and mark all freshly allocated memories with the current mark bit. Then, the freshly allocated memories will not be swept as they are marked, and the marking in the next round can be done correctly as it uses a different marking bit.

5.5 Performance Evaluation

In this section, we evaluate the experimental performance of the ActorNet platform. The focus of the experiments described here is to show that the overhead incurred by its constituent services is not prohibitive, and thus ActorNet is a suitable platform for mobile agents in resource-constrained sensor networks. First, we examine the page hit ratio of the virtual memory subsystem and its impact on system performance. We then evaluate the performance of the multi-phase garbage collector and finally describe the communication costs incurred by ActorNet.

5.5.1 Virtual Memory

We use the benchmark of computing the n^{th} Fibonacci number to evaluate the performance of the VM subsystem. A recursive version of this program is simple, but its exponential behavior is complex enough to carry out a performance evaluation.

As one might expect, as the page cache size increases, the page hit ratio increases. However, in a resource-limited computing environment such as a sensor node, we cannot increase the cache size indefinitely. We must consider a trade-off between the performance and the number of applications that can be run on the same platform (as not all applications use our VM). The first graph of Figure 5.7 shows the page hit ratio vs. cache size (number of pages in SRAM). Its shape is approximately concave and increases with cache size. After about 14 pages, the slope is almost flat. However, in the Mica2 platform, the flash write operations dominate the time spent in the VM subsystem. Hence, considering only the flash write operations as page-misses is a more accurate performance measure for the ActorNet platform. The second graph of

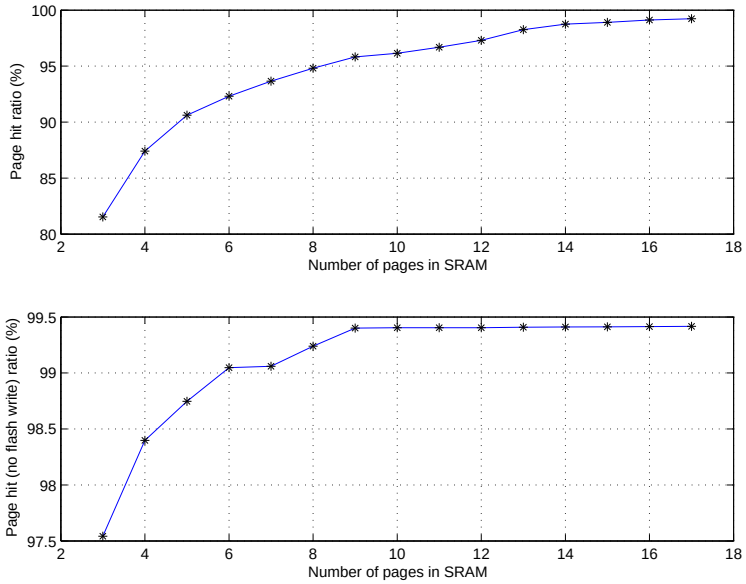


Figure 5.7: Page hit ratio (top) and non-flash-write page hit ratio (bottom).

Figure 5.7 shows the page hit ratio considering only the flash writes as a page miss. This graph shows a plateau after 9 cache pages (the current ActorNet implementation uses 8 cache pages). However, because of the `lock` count, when a message encoding or decoding task is running, it would use 7 cache pages. When there are 8 cache pages, the non-flash-write page hit ratio is 99.24%, while with 7 cache pages, the ratio becomes 99.06%.

5.5.2 Multi-Phase Garbage Collector

The slow flash write operation of the Mica2 poses a challenge for the garbage collection. As discussed earlier, the GC delay directly limits communication speed. In order to reduce the delay due to GC, we devised a multi-phase GC algorithm. We evaluate the performance of our multi-phase GC as a function of the number of pages swept per phase. The first graph of Figure 5.8 shows the number of flash write operations during a GC phase. The solid line shows an average number of flash writes, which can be interpreted as the expected delay due to GC for each phase, and the dashed line shows the maximum number of flash writes, which can be interpreted as the worst case GC delay per phase. The two lines are roughly increasing functions of the number of pages swept, which agrees with intuition. The second graph of Figure 5.8 shows the number of times GC is called during an experiment. As expected, it is a decreasing function of the number of pages swept per phase. The current implementation of ActorNet sweeps 10 pages per phase; its average number of flash write operations is 3.02 per GC. If we choose the number of pages swept to be 100, then the average number of flash writes is increased to 38.19. That is, when 10 pages are swept per phase, each GC phase

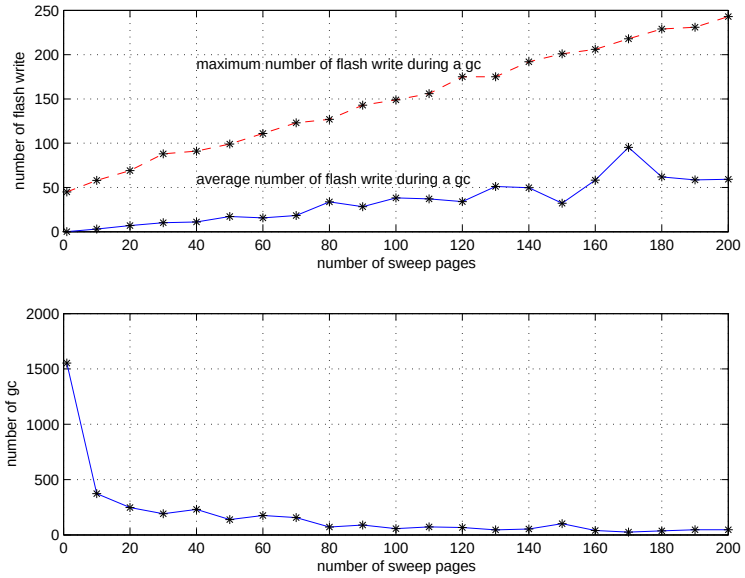


Figure 5.8: Number of flash writes during a GC phase (top) and number of GC phases (bottom).

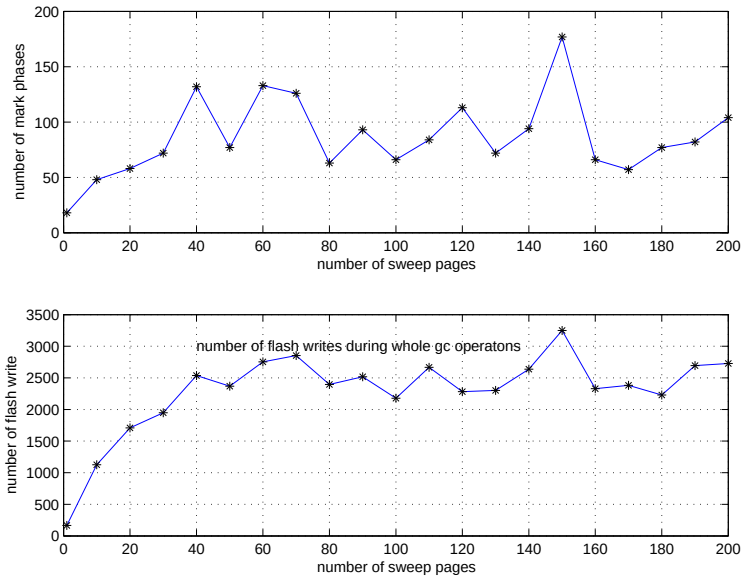


Figure 5.9: Number of mark phases (top) and number of flash writes due to GC (bottom).

message	content size (byte)	number of messages
`measure`	107	4
`temperature`	27	1
`move`	1629 + hop×8	57+
`return`	474 + hop×4	17+

Table 5.1: The number and size of messages transmitted by the actor program of Figure 5.1.

takes about 45.3 ms on average, and in the worst case it takes about 870 ms.

There is another merit of the multi-phase GC other than the reduced delay per GC phase. Because our memory reservation algorithm limits the search space for free memory within the interval of the last-swept pages, if the number of pages swept per phase is small, freshly allocated memory addresses are highly correlated in space and time. That is, the fewer the pages swept per phase, the higher the spatial and temporal locality of allocated data. The first graph of Figure 5.9 shows the number of mark operations during an experiment. Note that for each round of GC, there are a single mark phase and multiple sweep phases. Hence, the number of mark phases is an indicator of how efficiently the memory is used. The graph roughly shows that the number of GC rounds increases with the number of pages swept per phase. The second graph of Figure 5.9 shows the total number of flash writes made for GC during an experiment. Specifically, it shows an increasing, concave curve: when few pages are swept per phase, the related data tends to aggregate. Thus, related data is more likely to be found in the cache, which reduces the number of flash writes. However, sweeping too few pages at a time results in overly frequent calls to GC, as seen in the second graph of Figure 5.8.

5.5.3 Communication

Next we evaluate the communication costs of the example application of Section 5.2. This application does not require a routing service: it follows a steepest ascent path of temperatures, and also maintains a return path by itself. Also note that it does not involve spanning tree based data dissemination; the program migrates through the network, rather than collecting all the data at a central node. When the gradient path is a straight line, and assuming that the nodes are uniformly distributed, the number of nodes involved in the experiment is proportional to $\sqrt{n}$ for a WSN of n nodes.

To assess the communication performance, we measured the number and size of messages while running the actor program in Figure 5.1. Table 5.1 summarizes the results. Broadcasting a measurement actor to neighboring nodes requires 107 bytes of data in 4 messages. Sending a temperature reading requires 27 bytes, which can be sent in a single message. 27 bytes for a simple temperature reading may look like an overhead. The overhead can be attributed to the type information, the list data structure, and the communication

stack commands. However, they are necessary overheads to make actor messages generic and not application dependent. Observe that the measure actor is only 107 bytes long. A similar program that periodically samples the temperature and broadcasts the result is about 28 kB. In order to move an actor along the gradient ascent path, 1,629 bytes plus 8 bytes times the hop count thus far are necessary. The extra 8 bytes per hop account for the local variables stored during the migration (recursion). Note that as the actor migrates back to the base station, it discards the unnecessary pieces of its code. As such, the returning actor shrinks in size from 1,629+ bytes to 474+ bytes.

5.6 Discussion

We have developed ActorNet, a mobile agent platform for WSNs. ActorNet provides a uniform virtual computing environment that encompasses multiple WSNs, high-level abstractions for the actor coordination in distributed systems, and a set of essential features to develop agent systems on WSNs. The virtual memory and multi-tasking environments provided by ActorNet open the possibility for the more advanced coordination mechanisms such as tuple spaces [11], which can be built on top of the existing distributed storage services for sensor networks. Security is another concern for the mobile agent approaches. Mobile agent platforms can prevent malicious agents performing an admission control against signed agent programs. However, this security checking is more challenging in ActorNet because the program is mixed with the states and is changing over time. As a future research direction, we are researching whether the garbage collection can enhance the security. Despite these limitations, ActorNet provides powerful, efficient, scalable, and high level services for developing applications for WSNs.

Chapter 6

Synchronized Virtual Sensing

A common temporal frame of reference is required to interpret the information obtained from the sensors, and the lack of a global clock in such networks is problematic for many WSN tasks. In this chapter, we present a versatile service for collecting precisely synchronized sensor data from WSNs [1].

Time synchronization is a process by which distributed local clocks can agree to a consistent common global timeframe. Time synchronization services such as NTP [61] have long been used to synchronize computers over the Internet. In deeply embedded cyber-physical systems with non real-time architectures, however, time synchronization alone is insufficient. The combination of relatively high sampling rates (hundreds of Hz), limited processing capabilities and software-driven data acquisition makes synchronized clocks only a precondition for *synchronized data*.

We show that time synchronization must be combined with synchronized sensing in order to obtain useful, consistent data from distributed sensor nodes. This is particularly important for high data rate, data-intensive applications such as structural health monitoring, where data is generated too fast to be used immediately without offline reprocessing. We implement a different approach, called *synchronized virtual sensing*, which combines high-precision time synchronization with after-the-fact resampling of sensor data.

Specifically, we develop a precise time synchronization service based on the FTSP [56] algorithm, and extend it with individual sample timestamping and resampling middleware service. Based on experiments with acceleration measurement and subsequent signal processing, we show that synchronized sensing can achieve data quality sufficient for demanding WSN tasks such as vibration-based structural health monitoring.

The primary contributions of this chapter are: (1) introducing the concept of *data synchronization error*, as distinct from time synchronization error, for evaluating synchrony in WSN systems, (2) a resampling-based synchronized virtual sampling algorithm, and (3) experimental confirmation of the approach through application to a demanding structural health monitoring application. The service presented in this chapter reduces energy consumption more than 10-fold by reducing the required synchronization update frequency, while reducing the maximum synchronization error within a sensor network to under 100 μs. The middleware

[1] Parts of this work appeared in a previous publication [64].

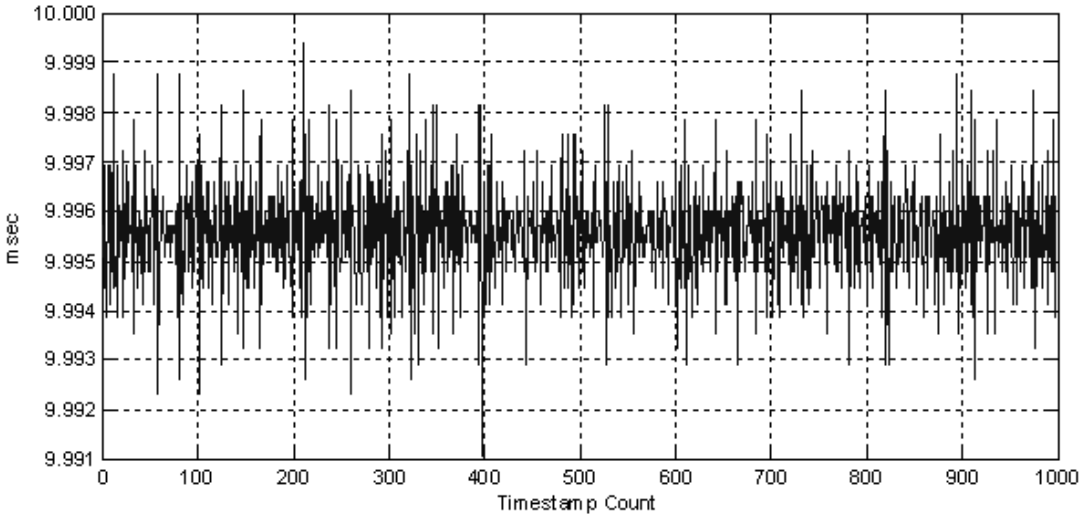


Figure 6.1: Time elapsed between timestamps taken every 10 samples.

service for obtaining highly precise synchronous data from a distributed sensing system has been developed and released to the community as open-source software.

Consider as an example a common class of vibration-based SHM algorithms rely on examining the *mode shapes* (characteristic vibration frequencies) of a structure. Small shifts in these frequencies may indicate structural problems. Detecting the *phase difference* in vibration from different parts of the structure, with acceleration sampled at 100 Hz, even 100 μs of synchronization error can result in a phase offset of 3.6 degrees, enough to produce erroneous output (false positives or false negatives) in the damage detection algorithm. Some SHM algorithms require sampling at frequencies of 500–1000Hz, exacerbating this problem even further. Thus highly precise synchronization is required for distributed sensor data to be of sufficient quality for use in SHM applications.

Two kinds of synchronization are needed to produce distributed sensor data of sufficient quality: clock synchronization and data synchronization. Clock synchronization assures that the different sensor nodes' clocks use the same basis for timestamping the sensor data. This alone is not sufficient: individual data samples must also be acquired *at the same exact time* across different nodes. Having two samples, accurately timestamped but taken at different times, does not provide an accurate basis for comparison of data across nodes. Data synchronization is needed to assure that samples are acquired at distributed sensors precisely on time. If such guarantees are impossible due to lack of end-to-end real-time control in the sensor network, data analysis and processing can be used to estimate the sensor values at the common time point.

In the next sections we address these two issues in turn. First we analyze and quantify synchronization errors between the clocks of distributed sensor nodes, and design an energy-efficient time synchronization

algorithm able to provide sufficient precision for use in SHM damage detection algorithms. Second, we address data synchronization, identifying several sources of error during to sampling.

6.1 Time Synchronization

Time synchronization is used in distributed systems to maintain synchrony between the clocks of each processor in the system. Even if all clocks are initially synchronized, minute differences in the rates of different clocks accumulate over time, resulting in *clock drift*. This necessitates periodic updating of local clock values to maintain a common view of global time, and possibly to maintain consistency with real time or an external clock. This process is called time synchronization. Time synchronization service enables the WSN application to construct a consistent global view of time across all nodes comprising the system. Precise time synchronization is critical for many sensor network applications, where data from an array of sensors is combined to obtain a global view of the physical environment. Unsynchronized sensor readings may lead to inconsistencies in that view.

High data rate WSN applications, such as vibration-based structural health monitoring, often require sampling rates in excess of 200 Hz, compared to more traditional sensor network applications such as environment monitoring and rare event detection [55, 19] with sampling periods on the order of minutes or even hours. Additionally, phase offsets play an important part in damage detection algorithms, in some cases requiring network-wide synchronization error between sensors to be under 100 microseconds. This level of precision is quite challenging to achieve in a multi-hop sensor network, as non-determinism in radio, clock and processor hardware of the sensor nodes results in errors averaging 20 microseconds *per hop*.

We develop an efficient method to overcome this problem, adaptively adjusting the time synchronization service to reduce clock skew and jitter to acceptable levels. Complex WSN applications may encompasses multiple modes of operation requiring different levels of precision. Short periods of intense sensing activity, where high-precision synchronization is an implicit requirement, may alternate with coordination, data aggregation and processing phases allowing for much looser synchronization.

Always using a time synchronization with maximum precision is impractical and wasteful in terms of energy usage. Our customizable time synchronization service is able to trade off precision and reduced resource consumption to best meet application demands at any point in time, so that the sensor network can control the quality of service (precision) of time synchronization according to its needs at each step of the algorithm and the amount of available network resources. Once the desired level of precision is achieved, our synchronization algorithm minimizes the overhead of maintaining synchrony by reducing the frequency of

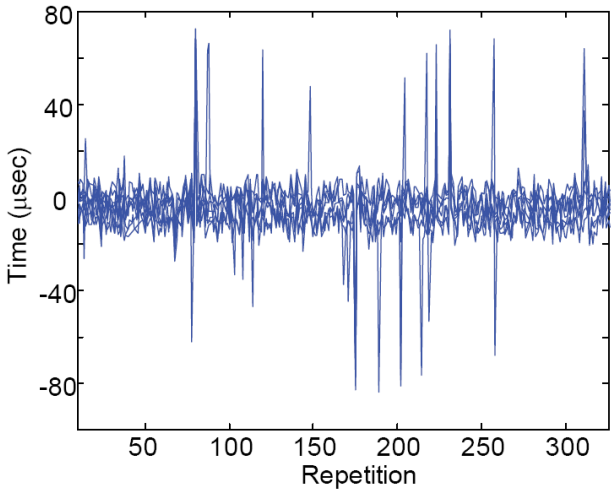


Figure 6.2: Instantaneous time synchronization error estimation.

updates. Minimum update frequency is determined based on the desired level of precision and the estimated clock drift with respect to the reference node.

6.1.1 Estimating Synchronization Error

Our time synchronization service is based on the Flooding Time Synchronization Protocol (FTSP) [56] implementation on the Imote2 platform. The algorithm utilizes time stamps on messages by the sender and receivers. A beacon node broadcasts a packet to all other nodes in range. At the end of the packet, a time stamp, t_{send}, is appended just before transmission. Upon reception of the packet, the receivers stamp the time, $t_{receive}$, using their own local clocks. The delivery time, $t_{delivery}$, between these two events includes interrupt handling time, encoding time, and decoding time. $t_{delivery}$ is usually not small enough to be ignored; however, its variance over time is usually small. $t_{delivery}$ can be estimated in several ways. An oscilloscope connected to both nodes and on-board clocks can keep track of the communication time stamp. In this study, $t_{delivery}$ is first assumed to be zero and then adjusted so that Imote2's placed on the same shake table produce synchronized acceleration data. If these nodes are synchronized, the phase offsets of the measured signals will be consistent over a wide range of frequencies. We find $t_{delivery}$ that gives a constant phase of zero. Using this value, the offset between the local clock on the receiver and the reference clock on the sender is determined as $t_{receive} - t_{send} - t_{delivery}$. This offset is subtracted from the local time when global time stamps are needed.

To evaluate time synchronization error, a group of nine Imote2's are programmed as follows. The beacon node transmits a beacon signal every 4 seconds. The other eight nodes estimate the global time using the

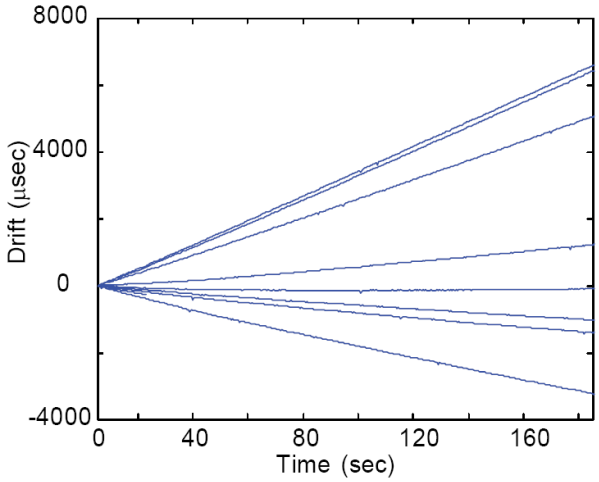


Figure 6.3: Uncompensated drift estimation.

beacon packet, thus performing time synchronization. Two seconds after the beacon signal, the beacon node sends the second packet, requesting replies. The receivers get time stamps on reception of this packet and convert them to a global time stamp using the offset estimated 2 seconds prior. These timestamps are subject to two error sources: First, time synchronization error, and second, delay in time stamping upon reception of the second packet. The receivers take turns to report back these time stamps.

The time stamps from the eight nodes are compared. Figure 6.2 shows the difference in the reported global time stamps using one of the eight nodes as a reference. The time synchronization error is under 10 microseconds most of the time. Scattered peaks indicate large instantaneous synchronization error. Note that the time synchronization error is one of the two above-mentioned factors explaining the error in the time stamps. This figure indicates that an upper-bound estimate of time synchronization error is about 80 microseconds.

Since such large errors are relatively rare, a median filter is applied to eliminate the effect of outliers. The time synchronization error thus modified is considered sufficiently small for precision-sensitive WSN applications such as structural health monitoring. A synchronization error of 10 microseconds corresponds to a 0.072-degree phase error for a signal at 20 Hz (typical vibration mode frequency), and the phase error is only 0.36 degrees at 100 Hz (maximum expected frequency).

The same approach is utilized to estimate clock drift. Upon receipt of the subsequent packet, the receivers return to the sender their offsets to estimate the global time, instead of global time stamps. If clocks on nodes are ticking at exactly the same rate, the offsets should be constant. In our experiments, however, the results did not show constant offsets. Figure 6.3 shows the offsets of eight receiver nodes. This figure shows that the drift rate is virtually constant over time. The maximum clock drift rate among this set of

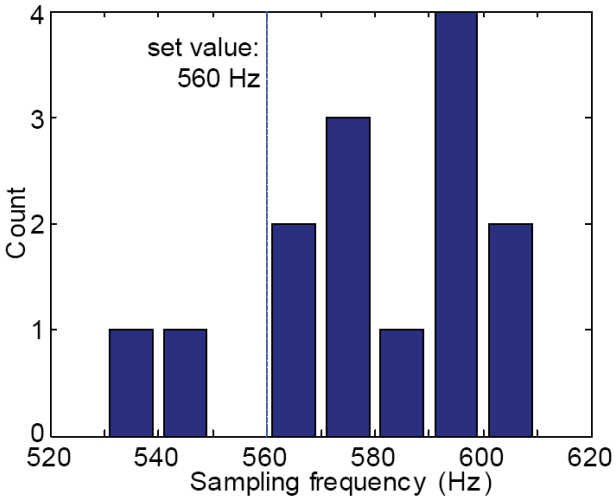


Figure 6.4: Significant unit-to-unit variation in sampling rates.

Imote2 nodes is estimated to be approximately 50 microseconds per second. Note that this estimate from the eight nodes is not the upper limit of clock drift because of the small sample size. This drift is small, but not negligible if long measurement records are taken. For example, after a 200 second measurement, the time synchronization error may become as large as 10 ms.

One solution to address this clock drift problem is frequent time synchronization. Time synchronization could be performed often to prevent the synchronization error from accumulating. However, frequent time synchronization is not always feasible, nor is it desirable from the energy consumption standpoint. For example, to maintain 100 μs maximum error bound in a network with clock drift up to 50 μs/s, it is necessary to resynchronize every 2 seconds. We take a different approach. The slope of the lines in Figure 6.3 are nearly constant and provide an estimate of the clock rate difference. If time synchronization offset values can be observed for some time, the slope can be estimated using a least-squares regression approach. The difference in clock rate is estimated and taken into account in the subsequent data processing as described in Section 6.2.

6.2 Synchronized Virtual Sensing

Measured signals from a smart sensor network with the intrinsic local time differences among the nodes need to be synchronized. As we have illustrated above, time synchronization among smart sensors does not necessarily offer synchronized measurement. Even when the clocks of sensor nodes are perfectly synchronized with each other, measured signals may not be accurately time-aligned. In this section, we investigate specific issues

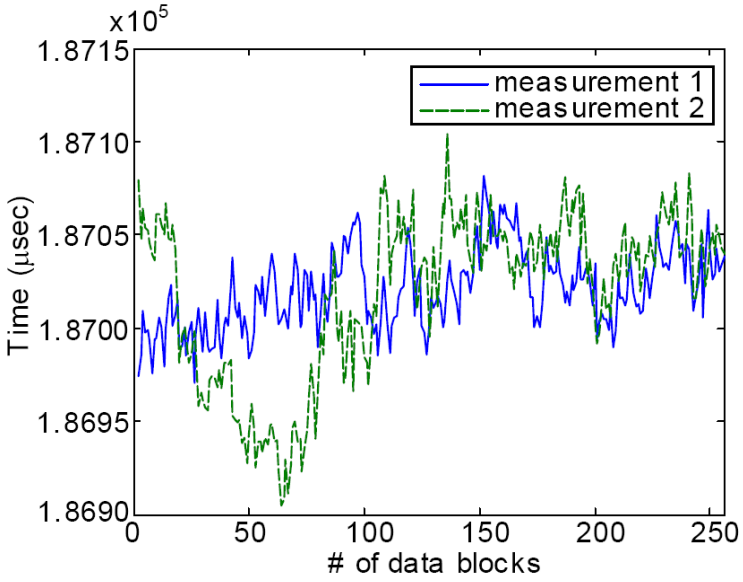


Figure 6.5: Sampling rate variance over time.

critical to synchronized sensing and implement synchronized sensing utilizing a resampling approach [62].

6.2.1 Problem Statement

Accurate synchronization of local clocks on Imote2's does not guarantee that measured signals are synchronized. Measurement timing cannot necessarily be controlled based on the global time. To better understand this situation, let us briefly consider what steps are involved in the Imote2 sensing process.

The sensing application on the Imote2 calls sensor driver commands to perform sensing and obtains measurement data in the following way. The application posts a task to prepare for sensing. Parameters such as the sampling frequency and the total number of data points are passed to the driver. Once the sensor driver becomes ready, data acquisition commences. The sensing task continues running until a predefined amount of data is acquired. During this process, the acquired data points are first stored in a buffer. Every time the buffer is filled, the driver passes the data to the sensing application. This block buffered data is supplied with a local timestamp, marked when the last data point of the block is acquired. The oscillator used for the time stamp runs at 3.25 MHz. If time synchronization is performed prior to sensing, the offset between the global and local times can be utilized to convert the local time stamp to a global time stamp when needed. The data and time stamps passed are copied to arrays in the sensing application, and the buffer is returned to the driver to be used for the next block of data.

Building synchronized sensing on this sampling process is challenging. The following difficulties may arise during distributed data acquisition, and need to be addressed by a synchronized sensing service.

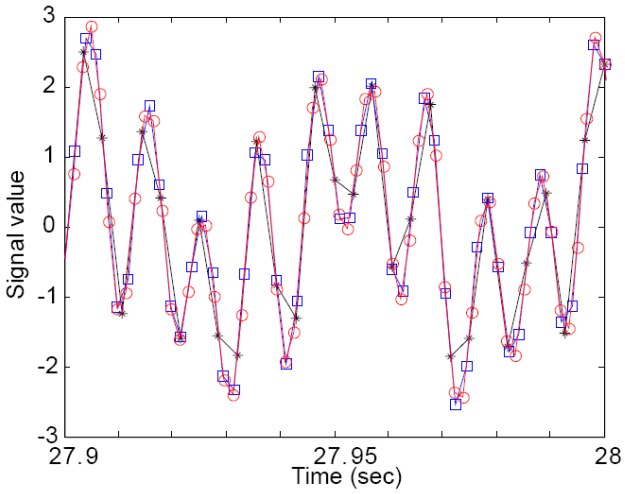


Figure 6.6: Measured signals before resampling.

Uncertainty in start-up time Starting sensing tasks at all of the Imote2 nodes in a synchronous manner is problematic. Even when the commands to start sensing are queued at exactly the same moment, the execution timing of the commands is different on each node. Thus, measured signals are not necessarily synchronized to each other.

TinyOS has only two types of threads of execution: tasks and hardware event handlers, leaving users little control to assign priority to commands; if sensing is queued as a task, this task is executed after all the tasks in the front of the queue are completed. As such, the waiting time cannot be predicted. If the command is invoked in a hardware interrupt as an asynchronous command, this command is executed immediately unless other hardware interrupts interfere. However, invocation of commands as a hardware interrupt from a clock firing at very high frequency is not practical; firing the timer at a frequency corresponding to the synchronization accuracy, tens of microseconds, is impossible.

In addition, the warm-up time for sensing devices after the invocation of the start command is not deterministic. Even if the commands are invoked at the same time, sensing will not start simultaneously.

Difference in sampling rate among sensor nodes The sampling frequency of the accelerometer on the available Imote2 sensor boards has non-negligible random error. According to the data sheet of the accelerometer, the sampling frequency may differ from the nominal value by at most 10 percent. Such variation was observed when 14 Imote2 sensor boards were calibrated on a shake table (see Figure 6.4). Differences in the sampling frequencies among the sensor nodes result in inaccurate estimation of structural properties unless appropriate post processing is performed. If signals from sensors with nonuniform sampling frequency are used for data analysis, incorrect frequency modes may be identified.

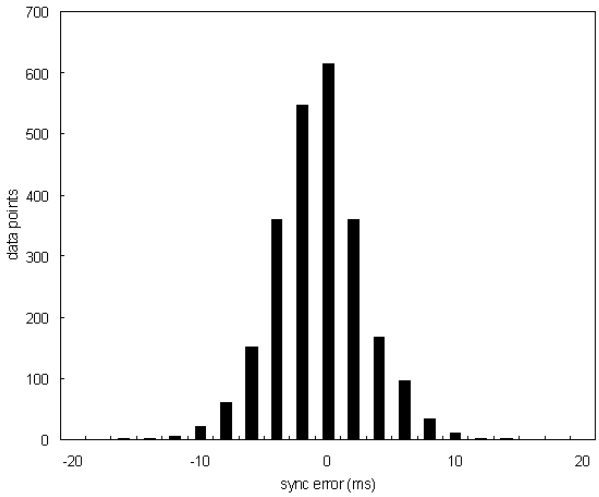


Figure 6.7: Histogram of measured data synchronization errors before resampling.

Fluctuation in sampling frequency over time Additionally, non-constant sampling rate was observed with the Imote2 sensor boards, which if not addressed, results in a seriously degraded acceleration measurement. When a block of data is available from the accelerometer, the Imote2 receives the digital acceleration signal and obtain the time stamp of the last data point. By comparing differences between two consecutive time stamps, the sampling frequency of the accelerometer is estimated. Figure 6.5 shows the difference between the time stamps when the block size is set at 110 data points. This figure provides an indication of the variation in the sampling frequency. The difference in two consecutive timestamps fluctuates by about 0.1 percent. Though imperfect time stamping on the Imote2 is a possible source of the apparent non-constant sampling rate, the slowly fluctuating trend suggests the variable sampling frequency as a credible cause of the phenomenon. With this fluctuation, measurement signals may suffer from a large synchronization error and a non-constant sampling rate.

6.2.2 Algorithm

Strict execution timing control is one possible solution for synchronized sensing. Real-time operating systems (RTOS) for industrial systems with ample hardware resources can manage command execution timing precisely. Small embedded systems without RTOS extensions can also be configured to manage execution timing precisely; however, implementation of real-time operation would make the system large and complex. Real-time control of wireless sensors in a network is particularly challenging; the situation is exacerbated for the high sampling rates required in many WSN applications. Also, even when a sensor node itself has real-time control, the peripheral devices such as sensor chip may have execution time delay or uncertainty

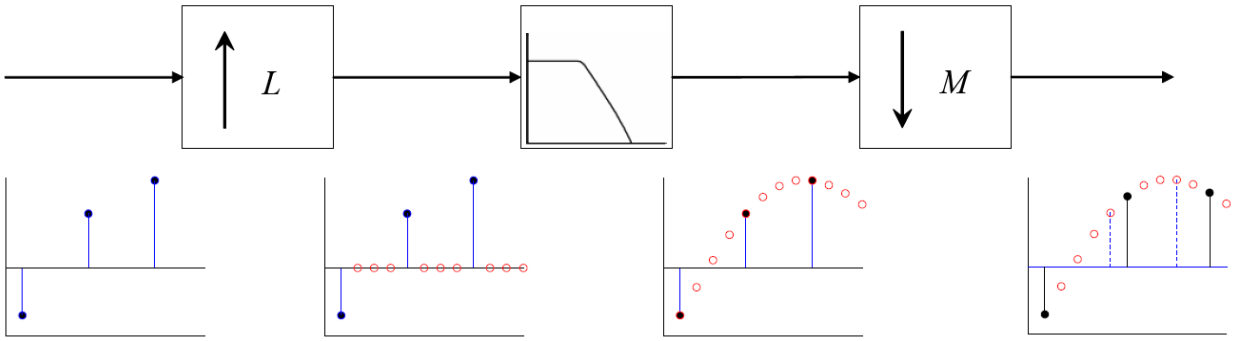


Figure 6.8: The resampling approach.

in timing. Instead of pursuing real time control, our synchronized sensing service is implemented using post-processing on the non-real-time OS of the Imote2 (TinyOS).

A "virtual sample" is computed on each sensor node, corresponding to the same global timestamp. Resampling based on the global time stamps addresses the three problems: (a) uncertainty in startup time, (b) difference in sampling rate among sensor nodes, and (c) time fluctuation in the sampling frequency. We first introduce the basics of resampling and polyphase implementation of resampling. The resampling approach is then modified to achieve a sampling rate conversion by a non-integer factor. Finally, this resampling method is combined with time stamps of measured data to address the three issues concurrently.

Resampling

Resampling by a rational factor is performed by a combination of interpolation, filtering, and decimation. Consider the case in which the ratio of the sampling rate before and after resampling is expressed as an rational factor, L/M. The signal is first up-sampled by a factor of L, then the signal is down-sampled by a factor of M. Before down-sampling, a low-pass filter is applied to eliminate aliasing and unnecessary high-frequency components (see Figure 6.8). Using this approach, the signal components below the filters cut-off frequency are kept unchanged through the resampling process, though imperfect filtering such as ripples in the pass band slightly distort the signal.

During up-sampling, the original signal $x[n]$ is interpolated with $L-1$ zeros as in Eq. 6.1,

$$v[n] = \begin{cases} x[n/L], & n = 0, \pm L, \pm 2L \\ 0, & \text{otherwise} \end{cases} \tag{6.1}$$

where $v[n]$ is the up-sampled signal. In the frequency domain, insertion of zeros creates scaled mirror images of the original signal in the frequency range between the new and old Nyquist frequencies. A

discrete-time low pass filter with a cutoff frequency, π/L, is applied so that all of these images except for the one corresponding to the original signal are eliminated. To scale properly, the gain of the filter is set to be L.

Before decimation, all of the frequency components above the new Nyquist frequency need to be eliminated. A discrete-time, low pass filter with a cutoff frequency π/M and gain 1 is applied. This low pass filter can be combined with the one in the up-sampling process. The cutoff frequency of the filter is set to be the smaller value π/L of and π/M. The gain is L. Decimation by a factor of M is then performed. Thus, the combination of up-sampling, filtering, and decimation completes the resampling process.

One of the possible error sources of this resampling process is imperfect filtering. A perfect filter, which has a unity gain in the pass band and a zero in the stop band, needs an infinite number of filter coefficients. With a finite number of filter coefficients, pass band and stop band ripples cannot be zero. A filter design with 0.1 to 2 percent ripple is frequently used. Figure 6.8 shows signals before and after resampling. A signal analytically defined as a combination of sinusoidal waves is sampled at three slightly different sampling frequencies. Two of the signals are then resampled at the sampling frequency of the other signal. As can be seen from Figure 6.8, after resampling, the three signals are almost identical. These signals are, however, not exactly the same due to the imperfect filtering. Though this signal distortion during filtering is preferably suppressed, this resampling process is not the only cause of such distortion. Other digital filters and AA filters also use imperfect filters. The filter in the resampling process needs to be designed so that it does not degrade signals as compared with the other filters.

Polyphase filtering

A Finite Impulse Response (FIR) filter can be computationally much less expensive than an Infinite Impulse Response (IIR) filter as a filter for resampling if the polyphase implementation is employed. The polyphase implementation leverages knowledge that up-sampling involves inserting many zeros and that an FIR filter does not need to calculate the output at all of the up-sampled data points. This implementation of resampling is explained in this section mainly in the time domain because the extension of the method to achieve synchronized sensing is pursued using a time domain analysis. Oppenheim et al. provided a detailed description of the polyphase implementation in the Z-domain [69].

The outputs do not need to be calculated at all of the data points of the up-sampled signal; rather, they should be calculated only at every M-th data point of the up-sampled signal. If an IIR filter were employed, the filter would need outputs at all of the data points of the up-sampled signal. Thus, this polyphase implementation reduces the number of numerical operations involved in resampling.

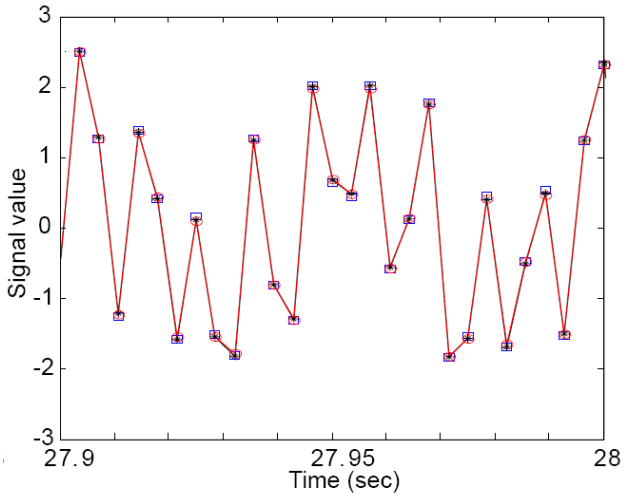


Figure 6.9: Measured signals after resampling.

6.2.3 Implementation

The resampling approach can be applied to achieve synchronized sensing [64]. The time stamp to indicate the beginning of the entire sensing process, T_0, and the predetermined waiting times, T_1, T_2, and T_3, are utilized to implement this resampling approach as well as the time stamp at the end of each block of data, $t_i (i = 1, 2, ..., n)$. T_1 is the waiting time before the sensing task is posted. A timer that fires every T_3 checks whether the waiting time T_1 has passed. T_2 is the waiting time before the Imote2 starts storing the measured data. The time stamps, t_i, are utilized later to compensate for misalignment of the starting time, as well as for individual differences in sampling frequency and the time varying sampling frequency. The waiting time T_4 is for sensing failure detection. If sensing has not finished at $T_0 + T_4$, the algorithm determines that the sensing process has failed and restarts the process.

When smart sensors are ready to begin sensing, a sensor node managing synchronized sensing in the sensor network gets a global time stamp, T_0, and multicasts it to the other network members. The predetermined waiting times, T_1 and T_2, are also multicast. All of the nodes then start a timer, which is set to be fired every T_3. When the timer is fired, the global time is checked. If the global time is larger than $T_0 + T_1$, a task to start sensing is posted, and the periodic timer stops firing. When a block of measured data becomes available, an event is signaled. In this event, the global time is checked again. If the global time is greater than $T_0 + T_2$, then the block of data is stored in memory. Otherwise, the block of data is discarded. From the time stamp of the last data point of the current block, t_{cur}, that of the last block, t_{last}, and $T_0 + T_2$, the time misalignment of the first data point of the block and the sensing start time are estimated. When the number

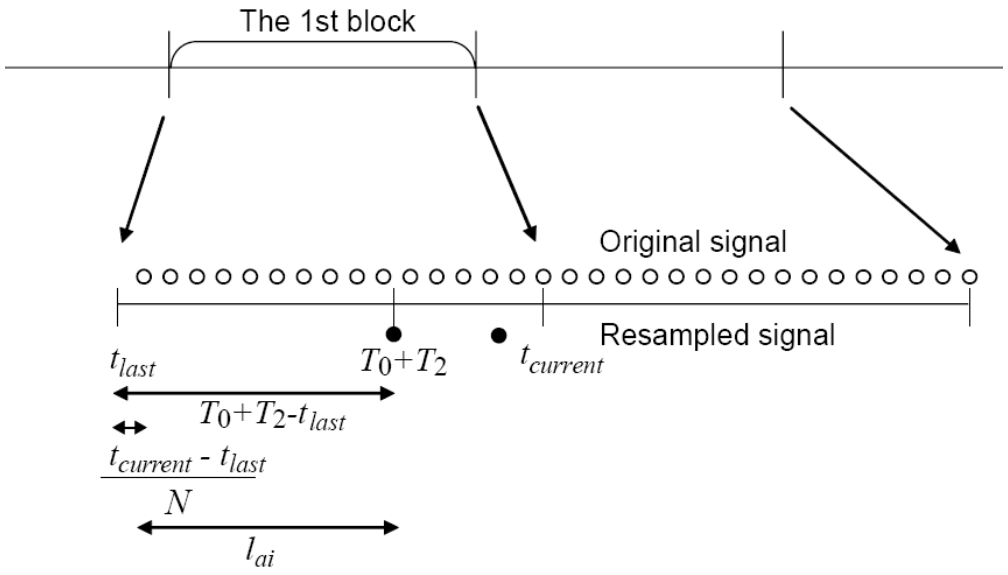


Figure 6.10: Resampling of the first block of data.

of data points in a block is N_{data}, the time misalignment of the first data point, t_{ini}, is estimated as follows:

$$t_{ini} = T_0 + T_2 - t_{last} - \frac{t_{cur} - t_{last}}{N_{data}} \tag{6.2}$$

where t_{ini} is later used in resampling as $l_{ai}$for the first block of data to compensate for misalignment of the starting time (see Figure 6.10).

Timestamps and resampling also compensate for difference and fluctuation in the sampling frequency (see Figure 6.5). The sampling frequency of the current data block, fs_{cur},is estimated from t_{cur} and t_{last}.

$$fs_{cur} = \frac{t_{cur} - t_{last}}{N_{data}} \tag{6.3}$$

The sampling rate conversion by a factor is applied to the block of data. The down-sampling factor is determined by

$$M_r = fs_{cur} \cdot L_a / fs \tag{6.4}$$

where fs is the sampling rate after the rate conversion. $l_{ai}$for the first block of data is t_{ini}. For the subsequent blocks, l_{ai} is determined by the time of the last resampled point of the previous block, $t_{last,resample}$. Because the calculation requires $x[m]$ in blocks before or after the current block, data in these blocks is also utilized. To be more specific, when resampling is applied to the current block of data, data from the block before and

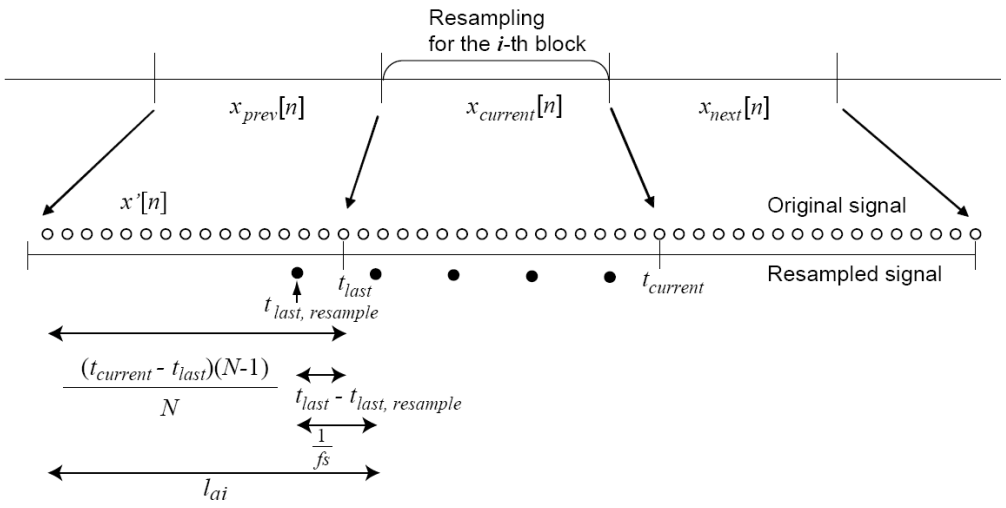


Figure 6.11: Resampling of the i-th block of data ($i > 1$).

after the current block is used as part of the input signal for the resampling, $x'[n]$, i.e.,

$$x'[n] = \begin{cases} x_{prev}[n] & 0 \le n \le N_{data} \\ x_{cur}[n - N_{data}] & N_{data} \le n < 2N_{data} \\ x_{next}[n - 2N_{data}] & 2N_{data} \le n < 3N_{data} \end{cases} \tag{6.5}$$

where x_{prev}, x_{cur}, and x_{next} are data in the previous, current, and next blocks, respectively. The sampling frequency of $x'[n]$ is assumed to be same as that of $x_{cur}[n]$. l_{ai} for $x'[n]$ is then calculated as

$$\begin{aligned} l_{ai} &= (t_{cur} - t_{last}) \cdot (N_{data} - 1)/N_{data} \\ &\quad -(t_{last} - t_{last,resample}) + \tfrac{1}{fs} \end{aligned} \tag{6.6}$$

This approach can thus address uncertainty in start-up time, differences in sampling rate among nodes, fluctuation of the frequency over time while the up-sampling factor L_a is kept moderate. However, it cannot be applied on the fly by the Imote2 nodes; resampling is applied after all of the Imote2's have acquired the data.

6.3 Discussion

Time synchronization and resampling-based synchronized virtual sensing middleware services for systems using smart sensors have been developed. The development realizes not only synchronized clocks, but truly synchronized data acquisition in distributed sensing applications, thus enabling highly accurate distributed

sensing for WSNs. After-the-fact resampling is suitable to high data rate applications, and is not tied deeply to particular features of operating systems or sensor boards. These middleware services are implemented for the Imote2 platform and have been released as an open-source library, and are expected to allow sensor network application developers to obtain reliable and accurate sensor information from dense deployments of smart sensors.

Synchronized virtual sensing has already had an impact in the domain of structural health monitoring, where distributed damage detection algorithms can now be implemented on WSNs, regardless of the quality of the sensors used. Additionally, the software approach to reducing sensing jitter and providing synchronized distributed sensor data allows sensor platform hardware, sensors, and operating systems to keep evolving independently, without coupling them through domain-specific application constraints.

Chapter 7

Ranging and Localization

Localization is the process of assigning location information to the nodes of a wireless sensor network. In general, *self-localization* is critical to the deployment of large-scale sensor networks, because manual surveying and entry of node coordinates is often impractical, and equipping each node with a specialized positioning device such as a GPS receiver is usually too costly, or does not provide sufficient accuracy.

While many algorithms have been proposed for self-localization of sensor networks, most have only been studied under simulation with idealized environmental conditions. Thus, in order to provide a reliable self-localization service to use in real environments conducive to ranging errors, we develop a localization algorithm suited to medium- to large-scale outdoor wireless sensor networks. This chapter describes these localization services, including results from experimental evaluation on an outdoor WSN deployment covering a 3300 m^2 area [1].

A ranging service is used to measure the distances between nodes, which are passed on as input to the localization service. The acoustic ranging service presented in this chapter is based on the time-difference of arrival method—distance is calculated based on the difference in propagation times of radio and acoustic signals. This technique allows us to achieve good accuracy and significant range using inexpensive, readily available speaker and microphone components.

Based on the experimental characteristics of our ranging service, we design a localization algorithm tolerant of sparse measurement data. Our localization scheme is based on *least squares scaling* (LSS) [15], a variant of the *multi-dimensional scaling* (MDS) technique. This comprehensive approach increases robustness by offering tolerance to the sparse and noisy measurements. We have further extended LSS with soft constraints to exploit deployment-specific requirements, such as minimum inter-node spacing. In combination, the above services can be used to provide accurate and reliable location information in a variety of environments and sensor network deployments that have proven challenging with other localization methods.

[1] Parts of this work appeared in a previous publication [44].

7.1 Long Distance Acoustic Ranging

In this section, we first introduce the basic acoustic ranging methodology and examine the sources of measurement error inherent to this medium. Then we introduce an advanced ranging service that can obtain measurements with higher accuracy and robustness, followed by an evaluation of results from field experiments.

7.1.1 TDoA Ranging

Our ranging service is based on the time difference of arrival (TDoA) between radio and acoustic signals, which utilizes the fact that these signals propagate at known but significantly different speeds: approximately 340 m/s for sound and almost instantaneously at short distances for radio waves. The TDoA method measures the arrival time difference between radio and acoustic signals originating at the same point to estimate the distances between nodes.

A bare-bones TDoA ranging service operates as follows. The sender broadcasts a radio message followed by an acoustic signal (chirp) with a known frequency signature. Each node receiving the radio message starts listening for the chirp. Detecting nodes then compute the difference in arrival times of the radio and acoustic signals, and consequently the distance. Unfortunately, a naive implementation of this method performs very poorly in the WSN setting.

Specifically, this simple approach suffers from low signal power, synchronization problems, multipath effects, and other complex issues from physical and digital domains. The sensor nodes' processing and storage capacities are tightly limited, as are the capabilities of inexpensive sensors and actuators typically found on such platforms. This in turn places significant limitations on the signal detection methods of the ranging service.

We thus have to consider several sources of error observed in TDoA ranging experiments, which have shown to be very significant: *(1)* Clock synchronization and timing effects, *(2)* Acoustic sensing and actuation delays, *(3)* Unit-to-unit variation, *(4)* Signal attenuation, *(5)* Environmental noise, *(6)* Multi-path propagation effects, and *(7)* Unreliable sensing. Some of these errors have very distinctive characteristics, which we can take into account when making distance estimates. E.g., we expect errors from sources 1, 2 and 3 follow a Gaussian distribution with a fairly small variance. Errors due to 4 and 5 are likely to be geographically correlated, whereas those from sources 6 and 7 may or may not be, depending on the situation.

7.1.2 Approach

Taking into consideration the sources of error listed above, we create a ranging service resilient against common types of errors. Range measurement accuracy is improved by employing clock synchronization, a sophisticated signal detection mechanism and by performing statistical filtering and consistency checking on the range estimates. We will now describe in some detail several components of our comprehensive approach to robust acoustic ranging, addressing and mitigating each source of error encountered in the original experiment.

Time synchronization Since the TDoA method relies on precisely measuring relatively short intervals over which sound travels between nodes, the clocks of source and destination nodes must be tightly synchronized to account for the delays incurred in transmission of the radio message. We synchronize source and destination nodes on an ad hoc basis using the same algorithm developed in Chapter 6.

Signal detection Our experiments indicate that the tone detector device on Mica2's MTS310 sensorboard is not be very reliable. In particular, we have observed that the probability of erroneous detection is strongly affected by environmental conditions at the time of measurement. Fortunately, the probability of detecting a tone in a sequence of measurements $b(t)$, $P[b(t) = 1]$, is much higher if a tone is actually present than if only background noise is.

Based on this model, we improve the confidence of acoustic signal detection by accumulating the binary outputs of the tone detector from multiple ranging attempts in a single buffer. Rather than using a single chirp, a series of chirps is generated by the sender node. The starting positions of multiple chirps between the two nodes are correlated; on the other hand, the random background noise triggering the detector is not. We apply *threshold detection* to make the decision: the sum of the samples must exceed the threshold value to be recorded as a true detection, and this must happen for a sufficient number of nearby samples for a chirp to be recognized. Figure 7.1 illustrates the detection mechanism with an example.

Robust detection To make the detector more robust, we encode a pattern in the acoustic signal. We use a sequence of identical chirps interspersed with intervals of silence. When detecting the signal, we look at both the chirp and the interval preceding it, allowing us to identify false detections due to noise or echoes that are not part of the pattern. To counteract the effect of echoes, we include small random delays between elements of the pattern. For identifying the tone itself, we apply threshold detection as described above. We

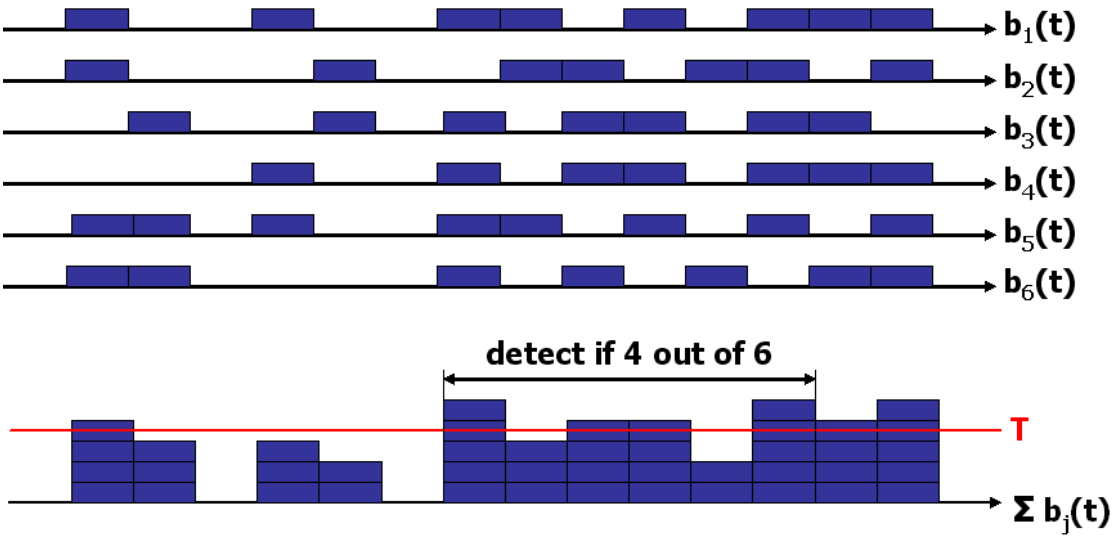


Figure 7.1: A simplified signal detection example with six chirps.

compute the detection time series detect(t) as

$$\text{detect}(t) = \delta_k \left[\sum_{i=0}^{n} \delta_T \left(\sum_{j=0}^{m} b_j(t-1) \right) \right]$$

where m is the number of samples accumulated, T is the threshold for signal detection, n is the total number of signal detections, k is the threshold for pattern identification, and $\delta_\theta(x) = 1$ if $x > \theta$ and 0 otherwise. The beginning of the acoustic signal is determined as the minimum value of t that satisfies detect(t) = 1.

Non-deterministic delays Finally, to compute the distance d_{ij} between source i and destination j, we need to take into account the various delays that take place in the ranging process: message transmission, speaker activation, detector (receiver) delay, etc.

We compute d_{ij} as follows. Denoting the non-deterministic message transmission delays at both sender and receiver as δ_{xmit}, we must introduce an additional constant time interval between the transmission of the radio message and the corresponding acoustic signal that is greater than δ_{xmit}. We denote this combined delay δ_{const}. The time difference of arrival can then be expressed as $t^j_{detect} - t^j_{send} - \delta_{const}$, where t^j_{send} is the time at which node i sent the radio message according to node j's clock. Since $t^j_{send} = t^j_{recv} - \delta_{xmit}$, we compute the distance using information locally available at node j as

$$d_{ij} = V_s \cdot \left(t^j_{detect} - (t^j_{recv} - \delta_{xmit}) - \delta_{const} \right)$$

where V_s is the estimate of the speed of sound.

Having addressed the above issues in the ranging method, we see significant improvement in accuracy over the basic ranging service outlined in the previous section.

Statistical filtering and consistency checking Even with the advanced signal detection algorithm, individual range estimates may still be erroneous, whether due to a low detection threshold, hardware malfunction, or some other cause. To guard against such problems, we make multiple distance measurements for a pair of nodes and filter the results to yield a more accurate estimate of the distance. Depending on the number of measurements available, we take the *median* or *mode* value of the measurements, which limits the effect of outliers. The mode operation is more resistant to the effects of uncorrelated large-magnitude errors than the median, but only if a sufficiently large number of measurements is available (otherwise, a value bucket with only two measurements may be the mode—hardly a reliable indicator).

Additionally, our ranging service employs inter-node consistency checks to identify measurements containing errors that may be correlated on a single node (e.g., errors due to faulty hardware or persistent wide-band noise). Bidirectional range estimates between a pair of nodes are discarded if they are inconsistent, and if three nodes have measurements between them, we can use the triangle inequality to identify the inconsistency.

7.1.3 Evaluation and Analysis

As in the baseline scenario, we evaluate the performance of the ranging service on a network of 46 Mica2 motes with the modified MTS310 sensor boards. We calibrate the ranging service in the target environment for best performance and determine the appropriate threshold values empirically. A high detection threshold is advantageous in noisy environments to limit false positives; on the other hand, a low threshold is more appropriate in high attenuation environments as it reduces false negatives.

Analysis: accuracy Figure 7.2 shows the error distribution for the distance measurements across all nodes after filtering and consistency checks. We can identify several distinct features of the error distribution from the figure. There is an approximately zero-mean, Gaussian component of the error with a small variance. This component is most likely due to timing effects, hardware delays and unit-to-unit variation. The fact that the error distribution is virtually zero-mean suggests increasing the number of samples and taking the median or mode is an effective technique for improving accuracy.

Another type of error is present in the measurements. These errors cluster to the right of the mean, caused by over-estimation due to late signal detections. The most likely explanation for these errors is that environmental effects (e.g., taller than average grass absorbing the signal more) causes more "misses" in

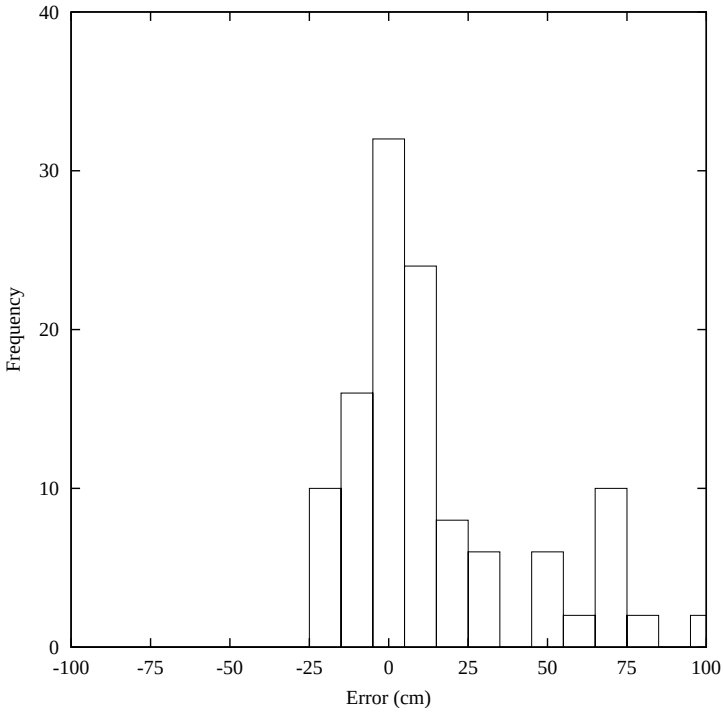


Figure 7.2: Ranging error histogram.

the detection of the early part of the signal. Since these errors are correlated with node positions, internal consistency checking will not be effective even if the number of measurements is increased.

Analysis: maximum range To determine the maximum detection range, we have tested the ranging service with a low detection threshold in quiet environments. While the maximum range varies with the features of the environment between each node pair, in our experiments it is 22m on 10–15 cm tall grass and over 35 m on pavement, on average. Interestingly, higher threshold values needed to make accurate measurements in noisy environments do not significantly reduce the maximum range (15–20 m on grass and 30 m on pavement).

By increasing the output signal power with a louder speaker and applying advanced filtering and consistency checking techniques, we have significantly extended the ranging service's measurement range in comparison to previous work, even in environments conducive to measurement errors. This represents a threefold improvement over previous work, while maintaining a distance-invariant median measurement error of about 1% of maximum range.

7.2 Least Squares Scaling Localization

To address the limitations of multilateration-based algorithms in environments with sparse range measurements, we develop a localization scheme based on *Least Squares Scaling* (LSS) [15] with soft constraints. In this section, we describe an LSS localization algorithm that is resilient against measurement errors and sparse ranging information common in low-density outdoor WSN deployments.

7.2.1 Algorithm

A node can compute a coordinate system of a group of nodes in a region if sufficient pairwise distance information between nodes is available. Given these local coordinate systems, we can compute an absolute coordinate system, where each node's position is based on external reference points, if more than three anchor nodes are available in the *entire* sensor field, so the number of anchors does not increase with the scale of the sensor network. The anchors' positions should be expressed in terms of the external coordinate system. However, even without a single anchor node we can still compute a relative coordinate system, where the positions of nodes respect the measured distances, but can be flipped, rotated and translated from some external reference system. This may be useful for example in geographic routing, where only relative node locations are important. Since finding a transformation, which is a combination the three operations, from a relative coordinate system to an absolute coordinate system accomplished easily, we will focus on finding a relative coordinate system.

Multidimensional scaling (MDS) is "any procedure which starts with the 'distance' between a set of points and finds a configuration of points, preferably, in a small number of dimensions, usually 2 or 3" [15]. Here, a configuration refers to a set of coordinate values. When distances between nodes are available, MDS finds their relative coordinates. In localization using classical MDS, the input distance matrix is transformed to a quadratic matrix of coordinates via double averaging. Then, the *singular value decomposition* (SVD) is applied to the quadratic matrix to calculate its principal components. The first two principal components are the configuration sought. One critical requirement is that distances between all pairs of nodes be known *a priori*.

An alternative technique is LSS [15], which seeks a configuration $C = \{(x_i, y_i) : i = 1, \ldots, n\}$ from a set of distances $D_{\text{full}} = \{d_{ij} : i, j = 1, \ldots, n\}$ by minimizing the unweighted error function E_u:

$$E_u = \sum_{d_{ij} \in D_{\text{full}}} (\hat{d}_{ij} - d_{ij})^2$$

where $\hat{d}_{ij} = \sqrt{(x_i - x_j)^2 + (y_i - y_j)^2}$, and d_{ij} is the measured distance between points (x_i, y_i) and (x_j, y_j).

An important property of LSS is that it still works using only a subset of D_{full}. This property allows LSS-based localization to tolerate sparse measurement data.

The error function is the sum of squares of differences between estimated distances and corresponding measured distances. As a result, errors in distance measurement are squared, too. Therefore, weighting distance measurements according to their confidence helps limit the effect of measurement errors on localization results. Statistical entities (e.g., standard deviation) can make a good choice for such weights. We extend the error function E_u to accommodate different weights by defining E_w:

$$E_w = \sum_{d_{ij} \in D} w_{ij} \cdot (\hat{d}_{ij} - d_{ij})^2$$

where $D \subseteq D_{\text{full}}$ is a set of distance measurements from a ranging service.

In many sensor deployments, a minimum distance between nodes can be known in advance: unless nodes are deployed via a purely random process, it is unlikely that two nodes will be placed very close together. Note that the minimum distance does not mean the *largest minimum separation distance* but a distance that is smaller than or equal to the largest minimum bound. Furthermore, our penalty-based constraint enforcement approach allows some nodes to violate the minimum separation condition, with locally distorted results. LSS allows us to incorporate this minimum spacing constraint into localization as a *soft constraint* [22]. Using the soft constraint, we penalize pairs of nodes which do not have distance measurements from the ranging service and whose assigned coordinates violate the minimum spacing constraint, so that any output solution would become more consistent. This can be visualized as straightening a plane which is incorrectly folded. Note that the set of penalized pairs dynamically changes as minimization progresses. With the soft constraint, the error function which we seek to minimize becomes:

$$E = E_w + \sum_{d_{ij} \notin D} w_D \cdot \left(\min\left(\hat{d}_{ij}, d_{min}\right) - d_{min} \right)^2$$

where d_{min} is the minimum node spacing and w_D is the weight for the soft constraint. Observe that $w_{ij} = 0$ for pairs of nodes which do not have distance measurements in D.

We use a gradient descent method to find a configuration that minimizes the error term. We can also use other methods such as *simulated annealing*. We update coordinates of the nodes at each time step using the rules

$$[\mathbf{x}^{t+1}, \mathbf{y}^{t+1}] = [\mathbf{x}^t, \mathbf{y}^t] - \alpha^t \cdot \nabla E|_{[\mathbf{x}^t, \mathbf{y}^t]}$$

where $\nabla E = \left[\frac{\partial E}{\partial x_1}, \ldots, \frac{\partial E}{\partial x_n}, \frac{\partial E}{\partial y_1}, \ldots, \frac{\partial E}{\partial y_n}\right]$ is the gradient of E and α^t is a step size that is either a small constant or a value that minimizes E along the line in the gradient direction at step t. Without the soft constraint,

$$\left.\frac{\partial E}{\partial x_i}\right|_{[\mathbf{x}^t,\mathbf{y}^t]} = \left.\frac{\partial E_u}{\partial x_i}\right|_{[\mathbf{x}^t,\mathbf{y}^t]} = w_{ij} \cdot \sum_{d_{ij} \in D} \frac{(\hat{d}^t_{ij} - d_{ij}) \cdot (x^t_i - x^t_j)}{\hat{d}^t_{ij}}$$

where $\hat{d}^t_{ij} = \sqrt{(x^t_i - x^t_j)^2 + (y^t_i - y^t_j)^2}$.
With the soft constraint,

$$\left.\frac{\partial E}{\partial x_i}\right|_{[\mathbf{x}^t,\mathbf{y}^t]} = \left.\frac{\partial E_u}{\partial x_i}\right|_{[\mathbf{x}^t,\mathbf{y}^t]} + w_D \cdot \sum_{d_{ij} \notin D} \begin{cases} \frac{(\hat{d}^t_{ij} - d_{min}) \cdot (x^t_i - x^t_j)}{\hat{d}^t_{ij}} & \text{if } \hat{d}^t_{ij} < d_{min} \\ 0 & \text{otherwise} \end{cases}$$

$\left.\frac{\partial E}{\partial y_i}\right|_{[\mathbf{x}^t,\mathbf{y}^t]}$ can be derived similarly.

In our experiments, we find an optimal step size α^t via a binary search along the gradient direction at each step. Note that this line search allows us to use a rather large step size, which results in better performance even though each step involves a binary search. For example, with a constant step size, the LSS algorithm converges with an α^t value as small as 0.001, but with line search it could be as large as 10, a huge improvement.

Our localization algorithm has two layers of minimization process. The bottom layer, `improve` method, refines the current position estimate by the gradient descent method and by a series of small perturbations, proportional to the current error level. This proportional perturbation is analogous to the random movement of the simulated annealing algorithm. That is, when the errors are large, the global configuration is distorted, which requires large change to correct, and when the errors are small, the local configurations are distorted for which small perturbation would speed up the convergence. `improve` method determines its arrival to the best point by counting the futile gradient descent minimization attempts. That is, if the current position doesn't improve for certain number of trials it returns.

The small perturbation improves the current position estimate efficiently, however it cannot overcome large local minima. To overcome this problem, the top layer, `explore` method, is introduced. `explore` method makes a series of large perturbation to the current best position estimate to initiate new paths of gradient descent based position improvements.

Algorithm 7.1 outlines the iterative localization algorithm. In `improve` the perturbation scale, `per`, is determined by scaling the current error level, `minErr`, by `Scale=0.01` on each loop. However to make the

Algorithm 7.1 Two level iterative localization algorithm.

```
double improve(double *Y) {
  double X[N*2], err, per, minErr = MaxErr;
  int loop, flat = 0;

  for (loop = 0; loop < Max1; loop++) {
    //add small perturbation
    per = max(MinPer, min(MaxPer, minErr * Scale));
    for (i=0; i<N*2; i++)
      X[i] = Y[i] + (rand() - 0.5)*per;

    //minimize by the gradient descent method
    err = minimize(X);

    //update position and reset the flat count
    if (err < minErr) {
      minErr= err
      for (i=0; i<N*2; i++)
        Y[i] = X[i];
      flat = 0;
    }
    else {
      //the number of continuous futile attempts
      flat = flat + 1;
      if(flat ≥ MaxFlat)
        return;
    }
  }
}
```

```
double explore(double *Z) {
  double Y[N*2], err, minErr=MaxErr;

  for (loop = 0; loop < Max2; loop++) {
    //add large perturbation
    for (i=0; i<N*2; i++)
      Y[i] = Z[i] + (rand() - 0.5)*ExpPer;

    //minimize
    err = improve(Y);

    //update the best position
    if (err < minErr) {
      minErr= err
      for (i=0; i<N*2; i++)
        Z[i] = Y[i];
    }
  }
}

void localize() {
  //N is the number of nodes and
  //Z is an array of x, y coordinates
  double Z[N*2] = {0,};
  explore(Z);
}
```

search effectively overcome the local minima and prevent it from searching blindly, we set the perturbation level within the bound `MaxPer=100` and `MinPer=10`. `improve` returns when `MaxFlat=100` consecutive minimization attempts become futile. `explore` tries to search different paths by perturbing the current best position by `ExpPer=100`.

7.2.2 Experimental Evaluation

To evaluate the resilience of LSS localization, we perform an experiment with 45 Mica2 motes equipped with standard sensor boards and our custom loudspeakers in a $60\times55\,\mathrm{m}^2$ grassy area, using the grid layout of Figure 7.3. The minimum inter-node spacing is 9.14 m. To allow a margin of error, we applied a soft constraint, with $d_{min} = 8.5m$, $w_{ij} = 1$, and $w_D = 10$. As w_D is increased, the rate of convergence increases similarly.

Figure 7.4 compares actual and estimated node positions. The computed coordinate system is translated and rotated to line up with the actual node coordinates. Most errors are found in the bottom left quadrant; the overall average localization error is 2.47 m. Without the largest five errors, the average improves to 1.5 m. As seen in Figure 7.3, lack of measured distances allows the two nodes in the (0∼10,10∼20) area to be

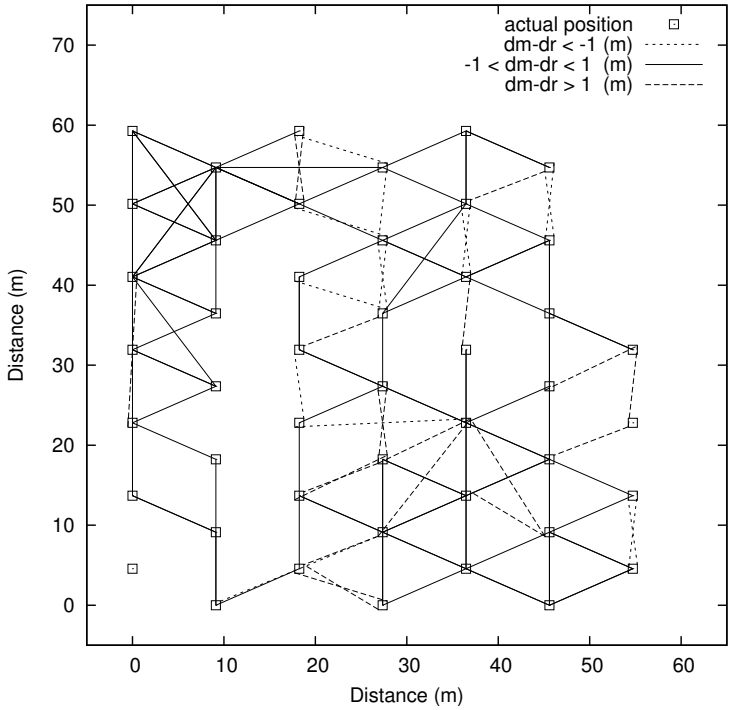


Figure 7.3: Sensor layout and pairwise measurements gathered via acoustic ranging.

swapped.

7.2.3 Effect of Node Layout

In the previous section, we evaluate the performance of LSS on a WSN of grid layout. Some of the reasons we choose the grid layout are to ease deploy the sensors at known positions for later analysis and to help evaluate the localization results systematically. However, the effect of this layout on the LSS localization problem must be addressed. To this end, we compare the results of the LSS method with an MDS variant, and apply the two algorithms to a simulated random deployment pattern.

First, for comparison purposes, we evaluate a Stress Majorization-based MDS algorithm on the same dataset. Unlike classical MDS, this algorithm does not require the full set of distance measurements, and the stress majorization is known to be good for preserving the global configuration. However, it fails to converge to the real configuration given the measured distances of Figure 7.3 (the average position error was 22.07 m). One observation we can glean from the result is that the estimated positions are not spread enough. We investigate the cause of this behavior by considering a random node configuration via simulation.

For the simulation, we uniformly place 50 nodes on a $50 \times 50\,\mathrm{m}^2$ area and generate distance measurements between nodes sampled from the normal distribution $N(d, 0.05 \cdot d)$, where d is the real distance between nodes.

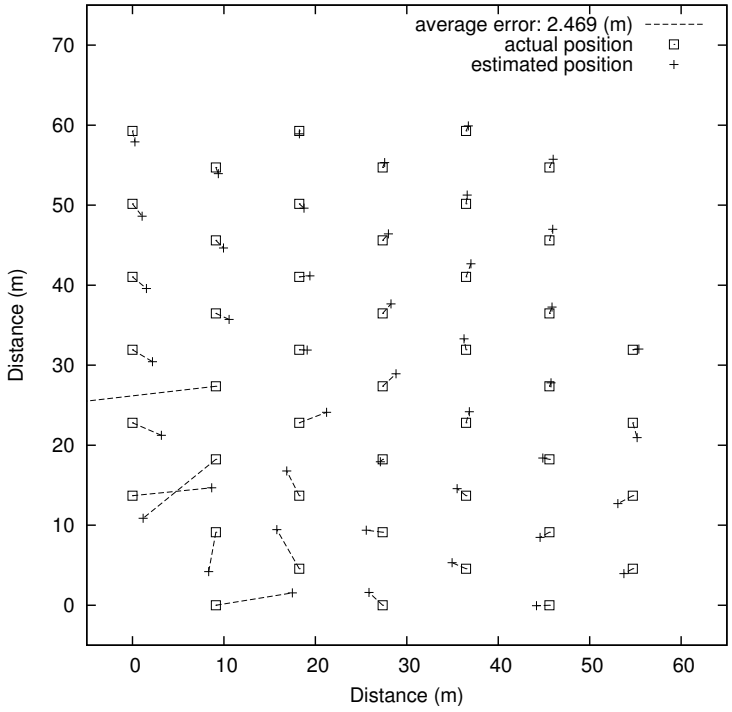
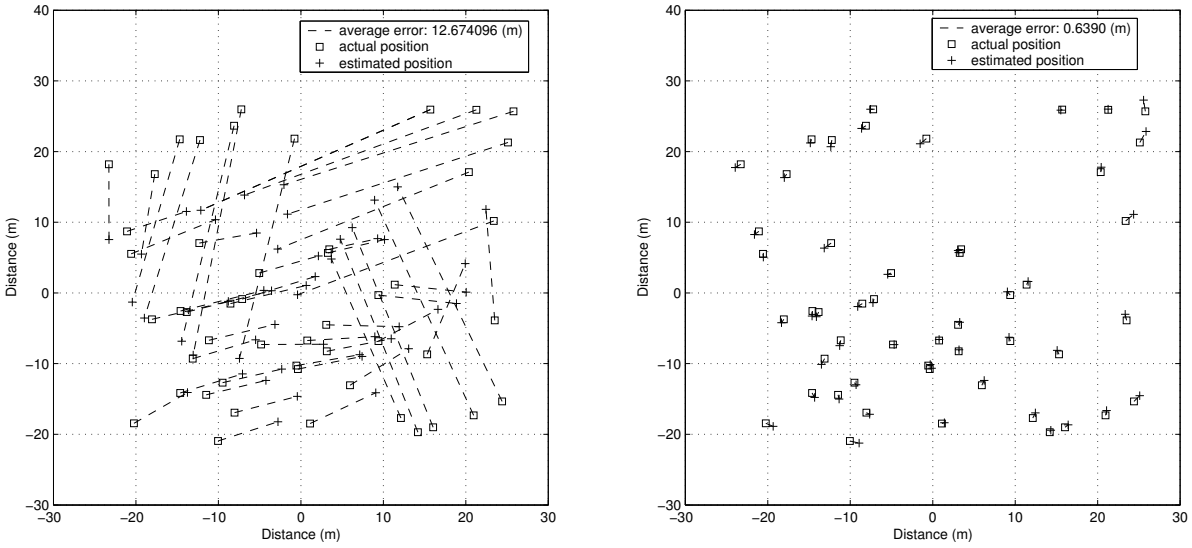


Figure 7.4: Result of LSS localization with a minimum spacing constraint.

Simulated distance measurements are created to match the characteristics of the actual deployment. To increase realism, we restrict the longest distance measurements to 25 m. Considering the random layout, we disregard the soft constraint on the minimum node separation distance by setting $w_0 = 0$ for this experiment.

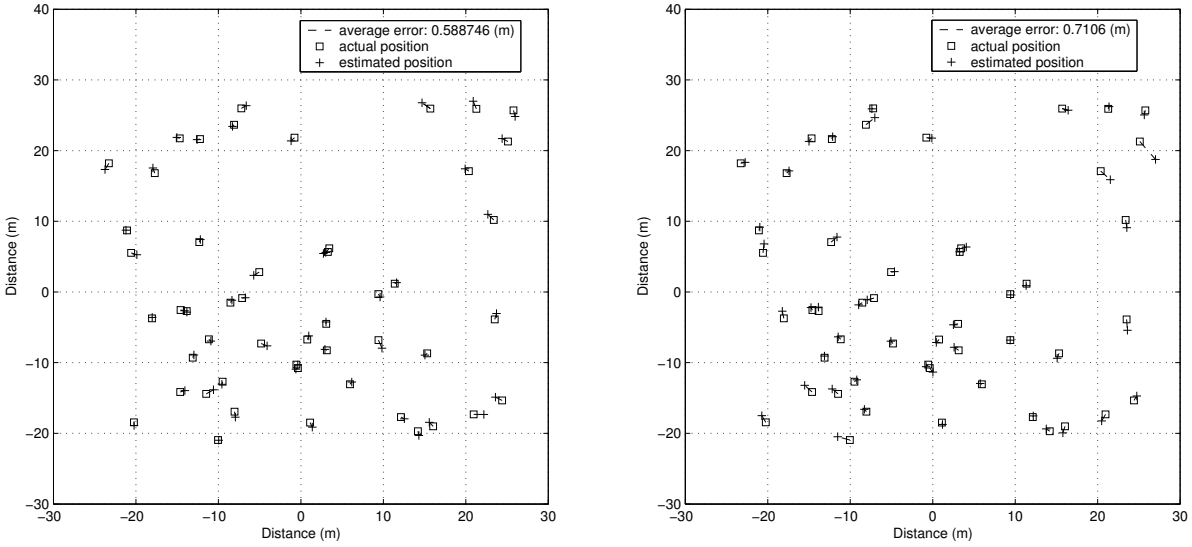
Figure 7.5 (b) shows the LSS localization result on the random layout. It has median and mean localization error of 0.590 m and 0.639 m. Large portions of the error can be attributed to the nodes at the top-right corner of the graph. Because these nodes do not have many measurements to the rest of the nodes, their configuration is slightly rotated from their real positions. However, apart from the global configuration, their position estimates are accurate relative to their local configuration.

Figure 7.5 (a) shows the result produced by MDS for the same node configuration. Again, many nodes fail to converge to the accurate position. After several experiments, we found that this MDS algorithm is not robust against the lack of long distance measurements. That is, the 25 m range restriction derived from experimental range measurements prevents the algorithm from converging to the real positions. In order to confirm this hypothesis, we repeat the MDS algorithm again on the same node configuration, but randomly remove the measured distances with a probability of 0.5 instead of systematically removing distances longer than 25 m. This results in 612 distance measurements available for localization, which is slightly larger than the previous experiment, where 600 measurements were used. Figure 7.6 (a) shows the result of the



(a) (b)

Figure 7.5: MDS (a) and LSS (b) when distances longer than 25 m are removed.



(a) (b)

Figure 7.6: MDS (a) and LSS (b) when 50% of the measured distances are randomly removed.

localization. The median and the mean localization errors of the experiment were 0.532 m and 0.678 m respectively. On the same condition, LSS algorithm produces a localization result with the median and mean errors of 0.555 m and 0.711 m respectively, which is essentially identical to the MDS result. Figure 7.6 (b) shows the LSS localization result.

From these results we can draw the conclusion that our LSS localization achieves the original design goal of robustness against sparse and inaccurate distance measurements, which are prevalent in actual WSN deployments. We feel that this result also underscores the importance of using actual measurement data, rather than pure simulations, in evaluating localization algorithms. Unrealistic simulation parameters, such as those used in Figures 7.6 (a) and 7.6 (b), fail to identify a critical flaw in the MDS-based approach.

7.3 Discussion

We have designed and implemented a fully functional localization system for wireless sensor networks, in particular for the Mica2 mote platform. The problems of sparse and noisy ranging measurements are addressed by the the holistic approach based on LSS localization. The addition of a minimum node spacing soft constraint to the LSS localization algorithm is successful in overcoming the problems of the measurements. Demonstrating good performance in realistic conditions, the system renders feasible deployment of large-scale autonomous WSNs in outdoor environments without prior surveying or manual configuration.

Several avenues of research remain to be explored. It may be possible to further improve the quality of ranging estimates, particularly through consistency checking, if additional sensing modalities are available to use in conjunction with acoustics. The gradient descent method of LSS localization algorithm has possibilities of improvement. It sometimes stuck at a deep local minimum and does not progress for a while. Such local minima often show locally correct but globally skewed results. Although, the consistent perturbation scheme will eventually overcome the barrier of the local minima, a more sophisticated approach might be necessary to overcome the sparse measurement problem and to speed up the localization process.

Chapter 8

Cooperative Object Tracking Application

Target tracking aims to detect the presence of an object and determine its path in an area of interest. This problem has been extensively studied in the field of signal processing. Tracking targets with geographically dispersed, cooperating sensors is attractive for several reasons. First, it can be more robust; sensors deployed close to targets would result in more reliable signal readings. Also, it can be more cost effective. A multitude of cheap sensors may track multiple targets simultaneously without human operators in the loop. However, the necessity for cost control and ensuing miniaturization limits the sensitivity of individual sensors and consequently the quality of sensor readings. The challenge is to design a tracking method which ingeniously reconciles the two defining characteristics, abundance in quantity and inferiority in quality, to realize the desired robustness.

We study the use of wireless sensors called *binary proximity sensors* in target tracking. These sensors provide only 1-bit information regarding a target's presence or absence in their vicinity. More specifically, we develop a tracking method in a two-tiered environment which consists of a self-configurable network of such sensors producing detection information and a computationally more capable node processing the information to estimate target positions.

Taken individually, outputs from binary sensors contain little information. The tracking method we develop co-opts past sensor outputs in addition to the current ones to improve the tracking resolution. The essence of the method lies in the fact that the trajectory of a target during a sufficiently small interval can often be approximated well by a straight line segment. A dynamic set of path points is maintained, which grows by the next path point which is estimated from the latest sensor outputs and shrinks if the current interval turns out to be too large. For the current set of path points, a best fitting line is sought and the current target position is computed from the line. This indirection has the positive effect of smoothing out errors inherent in outputs from simple binary sensors.

The target tracking application is implemented in a service-oriented fashion on the Mica2 sensor platform, incorporating the time synchronization and localization services described in the preceding chapters. The

application can coexist in the network and share these services with other service-based applications [1].

8.1 Tracking with Binary Sensors

In this section we develop a model of probabilistic binary sensors, an approximation model for a target's trajectory in a short time period, and a path-based tracking method based on the models, in turn. The essence of the path-based tracking lies in the way it correlates position estimates distributed in both space and time.

8.1.1 Modeling Binary-Sensor WSNs

We consider a network of N wireless binary sensors, each of which occupies a single point in a 2-D plane. Unlike sensors considered in traditional tracking approaches, binary sensors provide only one bit of data indicating presence or absence of a target in the sensing range. They are incapable of producing any other information, such as direction of arrival or distance to the source.

The two parameters that characterize the behavior of a sensor network are detection range and node density. We assume uniform node density and homogeneous detection ranges. The detection range of a sensor is a limit beyond which the sensor cannot distinguish a target's signal from ambient noise. Most research in this area assumes a uniform disc model for simplicity. However, detection ranges often vary widely among sensors depending on the sensitivity; fortunately much of individual variations can be compensated with careful calibration. A more critical factor is that detection ranges depend on the environmental conditions at the time of detection, such as the relative orientation of the object and the sensor. These factors make target detection near the boundary of the sensing range much less predictable.

8.1.2 Modeling Binary Proximity Sensors

The above observations give rise to a sensor model where a sensor detects presence of a target probabilistically near the boundary of its sensing range (Figure 8.1). The detection probability can be given as a function of the distance to the source. With the model, a sensor with a nominal sensing range R can always detect a target's presence if it is within $R - R_e$ range from the sensor. No signal from beyond distance R is ever detected. And, the detection probability drops off continuously as the distance increases between $R - R_e$ and R. We have empirically obtained this probability distribution from a prototype deployment for the simulation study in Section 8.2.2.

[1] Parts of this work appeared in previous publications [43, 59].

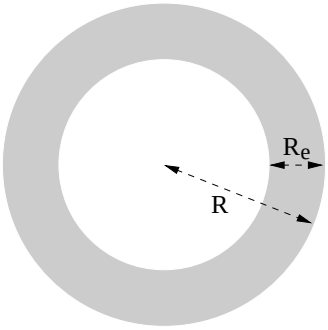


Figure 8.1: Probabilistic Sensor Model.

Algorithm 8.1 Path-based target tracking

1: Each node records detection durations.
2: Triples (time, location, duration) from the nodes are aggregated at the tracking node, which then computes
3: The weights for the detecting sensors and then path points.
4: The line which best fits the path points.
5: The target's position using the velocity and line estimates.

8.1.3 Modeling the Target Trajectory

Objects can move arbitrarily, possibly changing speed and direction at any time. Precise modeling of such arbitrary paths can prove cumbersome and unnecessarily complex for target tracking in binary-sensor WSNs. In lieu of costly precise path modeling, we adopt a piecewise linear approximation.

We allocate a small moving window to the past measurements and use a straight line segment to represent an object's trajectory in that window. Since a target's movement is governed by the laws of physics and the length of the window is determined by the network's sampling rate, a straight line segment can approximate a target's path fairly well for a short period of time, assuming that the sampling rate is reasonably high and that the target's movement is not completely erratic.

The extent to which the target's actual path diverges from its straight line approximation in a given window depends on several factors, including the target's speed and turning radius. For vehicles traveling along a highway, the difference can be very small. For a person walking along a curvy path with tight turns, the divergence may not be negligible. In either case, accuracy improves as we increase the resolution of the WSN, either by increasing the node density or by reducing the sensing range.

8.1.4 Tracking with Path Estimation

A straightforward way for position estimation using binary sensors is the centroid method; it estimates a target's position at a given time as the average of the detecting sensors' positions. It is attractive for its simplicity, but it produces estimates of subpar quality (Figure 8.5).

Instead of relying on direct methods which estimate a target's position from the detecting sensors' locations, we have adopted an indirect, two-step method. The basis of the approach remains the same as the centroid method; each detecting sensor approximates the target's position with its own location. However, the two-step method differs from the centroid method in the way it correlates the sensor data.

The method computes the weighted average of the detecting sensors' locations and uses it as an estimate of a point in the target's path. Then, it finds a line which best fits the point and other points from the recent past. (Recall that a target is assumed to have a straight line trajectory in the short term.) Finally, using the velocity estimate and the line equation, the target's position at the time is estimated. As evident from the evaluation results given in Section 8.2.2, indirection through path estimation tends to smooth out errors in the position estimates. The method is summarized in Algorithm 8.1.

A number of variations may emerge from this method skeleton depending on the weighting scheme. In fact, the performance of the path-based tracking is closely related to the chosen weighting scheme. The simplest scheme is to weight sensors uniformly, which effectively puts a target at the center of mass of the detecting sensors. However, it fails to take advantage of additional information such as the sensor's closeness to the target. We develop dynamic, history-based weighting schemes with a trade-off between computational cost and accuracy. The schemes are based on a simple relation between a target's proximity to a sensor and the sensor's detection duration.

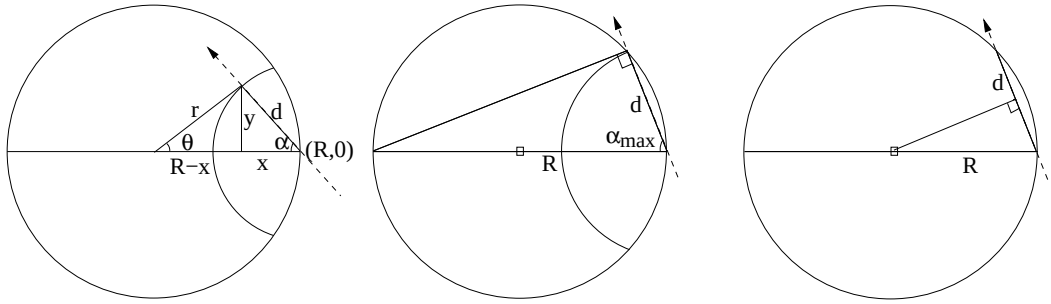
8.1.5 Distance-based Weighting

We seek to weight sensors in a manner that assigns a higher value to sensors close to the target. Consider a target moving in a sensor's range of detection (Fig. 8.2 (a)). Observe that a target generally stays longer in the sensing range of a sensor it passes closer than one farther. Thus, a quantity related to the duration for which the sensor has tracked the target would make a good candidate for such weights.

The detection duration gives an estimate of the distance the target has traveled inside the sensing range of the sensor. For simplicity, assume a sensor's detection range is a perfect circle, i.e., $R_e = 0$. For a target moving at a constant speed v, the distance $d = vt$ when the detection duration is t. We can relate the distance traveled within the sensor range to the current distance between the sensor and the object in several ways.

Pessimistic

This method assumes the target is on the boundary of the detecting sensor's sensing range at the time of detection and is about to move out of the range. Thus, it uses $1/R$ as the weight. The use of the successive



(a) target position after traveling d with entrance angle θ (b) maximum entrance angle α_{max} given distance d (c) distance to the path

Figure 8.2: Object location and path after traveling distance d within sensor range.

refinement described in Section 8.1.6 makes it differ from the "uniform" weighting method.

Expected Distance

Since a target can enter a sensor's sensing range from any point at any angle, it can be anywhere in the annulus defined by the outer circle of radius R and the inner circle of radius d. Without loss of generality, assume that the target enters the range at $(R, 0)$ (Figure 8.2 (a)). Then, given d, the relative direction angle α is bounded by α_{max} from above, which is $\cos^{-1}(d/2R)$ (Figure 8.2 (b)). The equation of the arc is given as

$$r = \sqrt{R^2 - 2Rd\cos\alpha + d^2},\ 0 \le d \le 2R,\ |\alpha| \le \cos^{-1}\frac{d}{2R}$$
$$\alpha = \cos^{-1}(\frac{R^2 + d^2 - r^2}{2Rd})$$

The cumulative distribution function $F(r) = P(X \le r)$ of random variable X indicating the probability that the distance to the target is less than or equal to r can be found by integrating the probability that the target enters the range with a certain direction angle over the region where α corresponds to the distance less than or equal to r.

$$\begin{aligned} F(r) &= P(X \le r) = \int_{-\alpha}^{\alpha} p(t)dt = 2\alpha\frac{1}{2\alpha_{max}} \\ &= \cos^{-1}\frac{R^2 + d^2 - r^2}{2Rd} / \cos^{-1}\frac{d}{2R} \end{aligned}$$

From the relation between d and $F(r)$ we compute the expected distance to the target as

$$r_e = \int_{r_{\min}}^{r_{\max}} r \cdot F_d(r)dr, r_{\min} = R - d, r_{\max} = R. \tag{8.1}$$

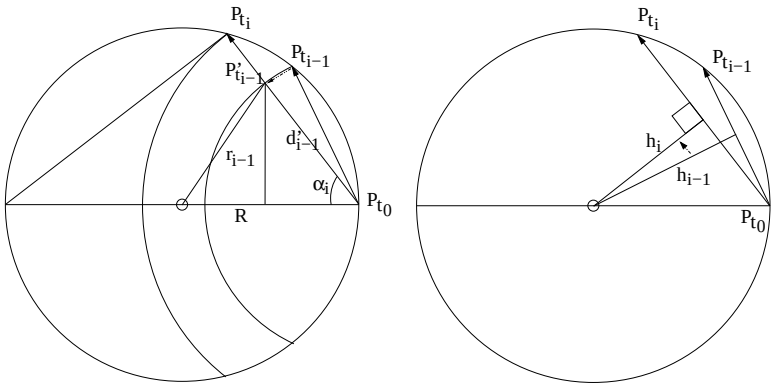


Figure 8.3: Successive refinement technique.

and use $\frac{1}{r_e}$ as the weight.

Distance to the Path

With the assumption that the current detection is the last (thus, *pessimistic*), we compute the distance to the path $h = \sqrt{R^2 - (d_i/2)^2}$ and use $\frac{1}{h}$ as the weight (Figure 8.2 (c)).

8.1.6 Successive Refinement

Given a target's straight line trajectory, the closer it passes to a sensor, the longer it stays in that sensor's detection range. The exact duration can only be known after the target moves beyond the sensor's range. This is the basis of the pessimistic assumption that the current detection is the last one in computing the weights. Consequently, duration measurements from an active sensor are most likely underestimates; another detection by the sensor rectifies the target has passed the sensor closer than estimated. Recall that we estimate points on the target's path and find a line equation to compute the current target position. The essential idea of successive refinement is to re-compute the past weights using the latest duration information to estimate the path points more precisely.

Pessimistic/Expected Distance Let d_i be the length of secant $\overline{P_{t_0}P_{t_i}}$ in Figure 8.3 (a). Since $\cos\alpha_i = \frac{d_i}{2R}$, for $0 < k < i$,

$r'^2_k = R^2 + d_k^2 - 2Rd_k\cos\alpha_i = R^2 + d_k^2 - d_k d_i$. For the Pessimistic scheme, the new weight at t_k is $\frac{1}{r'_k}$. For the Expected Distance scheme, use $r^{max} = r'_k$ in Eq. 8.1 to compute the expected distance at P'_k.

Distance to the Path The distance to the path at P_i is $h_i^2 = R^2 - (\frac{d_i}{2})^2$ and it is also the distance to the path at P'_k, $0 < k < i$ (Figure 8.3 (b)).

8.2 Performance Evaluation

We have evaluated the path-based method and compared its performance under different conditions via an extensive simulation study. To make evaluation results more convincing and useful, we have taken a realism-based approach where relevant parameters are collected from actual devices and fed into subsequent simulation runs. First, we have implemented a proof-of-concept acoustic tracking prototype using Mica2 motes and experimentally validated its performance. Microphones and tone detectors in standard Mica2 sensor boards are used as acoustic sensors. Although some acoustic sensors are capable of providing such information as signal strength and direction of arrival of the sound, we assume that the tone detectors do not provide such information, treating them as binary sensors. Then, we have instrumented the simulated sensors with the parameters we have collected from the prototype and compared the tracking performance under different configurations and conditions. For the simulation study, we assume that the nodes in a network are localized *a priori*, the clocks of the nodes are periodically synchronized, and an appropriate MAC protocol and a data aggregation scheme are in place, since issues such as routing and networking are not a primary focus of this study.

8.2.1 Object Tracking with Binary Acoustic Sensors

To verify the viability of path-based object tracking and to collect relevant parameters to model sensors in the subsequent simulation study, we have implemented the tracking method on a small-scale network of Mica2 motes [31] and conducted an experiment using acoustic sensors. The tone detector on the Mica2 sensorboard detects a specific range of sound frequencies and provides binary output indicating the presence or absence of a signal.

The network consists of 25 motes, laid out in a regular 5×5 grid. Nodes are placed 0.4 m (1 u) apart, and the gain on the microphone is adjusted to provide a reliable detection range of $R = 0.6$ m (1.5 u). An RC car equipped with a mote plays the role of a target. It moves through the network at the approximately constant speed of 0.2 m/sec which amounts to 0.5 u/sec. It should be noted that the boost power of each microphone as well as the speed of the object were significantly reduced in the laboratory experiment. With proper calibration, the same hardware can cover a wider area and track faster moving objects.

The results of this experiment are shown in Figure 8.4. No successive refinement is employed in this set of experiments. No ambient noise is artificially introduced, but echoes are abundant. The distance-to-the-path scheme was used due to the lack of hardware support for floating point operations in Mica2 motes. We observe the presence of significant errors at the initial period, which is attributed to the lack of reliable detection and velocity data; as more detections are recorded, errors quickly settle at a lower level.

Also shown in Figure 8.4 are the results from a simulation run with the sensor model proposed in Section 8.1.2 and with the parameters collected from the experiment (e.g., $R_e = 0.2R$.) The two results show comparable behaviors, although the magnitude of the errors is markedly higher in the actual experiment than in the simulation. This discrepancy can be attributed in part to the quality of the sensors. In the simulation, we assume ideal sensors which always detect a target within range $R - R_e$, and with diminishing probability beyond that range. Actual sensors are likely to have somewhat anisotropic detection ranges. Furthermore, echoes could have caused false detections. Tone detectors in Mica2 are extremely unreliable; they sometimes miss presence of a sound signal emitting from a nearby source.

8.2.2 Evaluation with Simulation

Given the encouraging experimental results, we have conducted an extensive simulation study and evaluated the performance of the path-based tracking method with the three weighting schemes.

A 900-sensor network with a single mobile object is simulated. Unless otherwise noted, results are obtained using 30x30 grid placement with the origin at the center. The target travels from the origin along the direction of a vector (20,11) at the speed of 1.2 $unit/sec$. The successive refinement is employed in all the simulation results reported in the section.

Baseline Performance

Figure 8.5 shows position estimation errors over time for the centroid method which computes the average of detecting sensors' locations as a target location. The plot labeled "indirect" has been obtained by first estimating path points using the centroid method, and then estimating the path using these points. The results suggest the validity of the weighting schemes proposed above.

Performance of Different Weighting Schemes

First we study performance of three weighting schemes in the path-based target tracking using the grid node placement noted above. Figure 8.6 plots changes in magnitudes of estimation errors [2] over time for the three schemes. We observe that the magnitudes decrease in all three weighting schemes as more data become available. These results show that the path-based tracking performs quite well. The Expected Distance schemes achieves the best results among the three, but it is also most computationally expensive to run. Distance-to-the-path takes a moderate amount of computation to yield good results. We attribute the good performance to the facts that indirection through path estimation using more accurate past estimates

[2]The errors are measured as difference between an estimated target position and the corresponding actual position.

Algorithm 8.2 Adaptive path-based target tracking

```
1: – 2: same as Algorithm 8.1.
3: {Let ω be the current window.}
4: ω := (ω < ω_min)?ω_min:((ω ≥ ω_max)?ω_max : ω)
5: loop
6:    Find a line that best fits the path points within w.
7:    Use the line to estimate the target's position p.
8:    if (is_good(p)) then ω = ω + 1; break; end if
9:    if (ω ≤ 0) then use a fall-back for p; break; end if
10:   ω := ω − Δω
11: end loop
```

smooths out errors in individual path points near the target's current position and compensates for scarcity of information in binary readings.

8.2.3 Adaptive Path-based Target Tracking

The path-based tracking is very effective for tracking targets with a straight line or near-linear trajectory. The results may well be comparable for targets moving along a line with a small curvature. But, it is hardly conceivable that this would work well for targets with a trajectory of a high curvature. To determine if more robust path-based tracking is feasible, we have extended the baseline method so that the window size for past detections can be dynamically adjusted (Algorithm 8.2). Two important parameters are when to forget (or when to stop forgetting) these points (i.e., is_good(p)) and how quickly to forget (i.e., $\Delta\omega$). For the experiments, we used a version of is_good(p) implemented using overlapping ranges of detecting sensors and $\Delta\omega = 1$. Note that $\Delta\omega$ can also be adjusted dynamically depending on the target's movement characteristics.

8.3 Discussion

Errors in path-based tracking with binary sensors come from multiple sources. Some are determined at the deployment time—e.g., placement and localization errors. Others are more dynamic; for example, sampling signals at discrete intervals makes the first and last detections of a sensor occur away from the exact boundary of the sensing range. These errors can be managed with some additional cost. On the other hand, some errors are inherent to the sensors considered in this study.

The tracking method presented in this chapter finds a straight line which approximates the path of a target during a short period of time, and uses the line to estimate the target's current position. Two design parameters of the method, sensing range and node density, have been evaluated through a combination of a prototypical implementation with acoustic sensors and an extensive simulation study. The results suggest that the path-based tracking is effective for binary sensor networks with a small sensing range and a high node density.

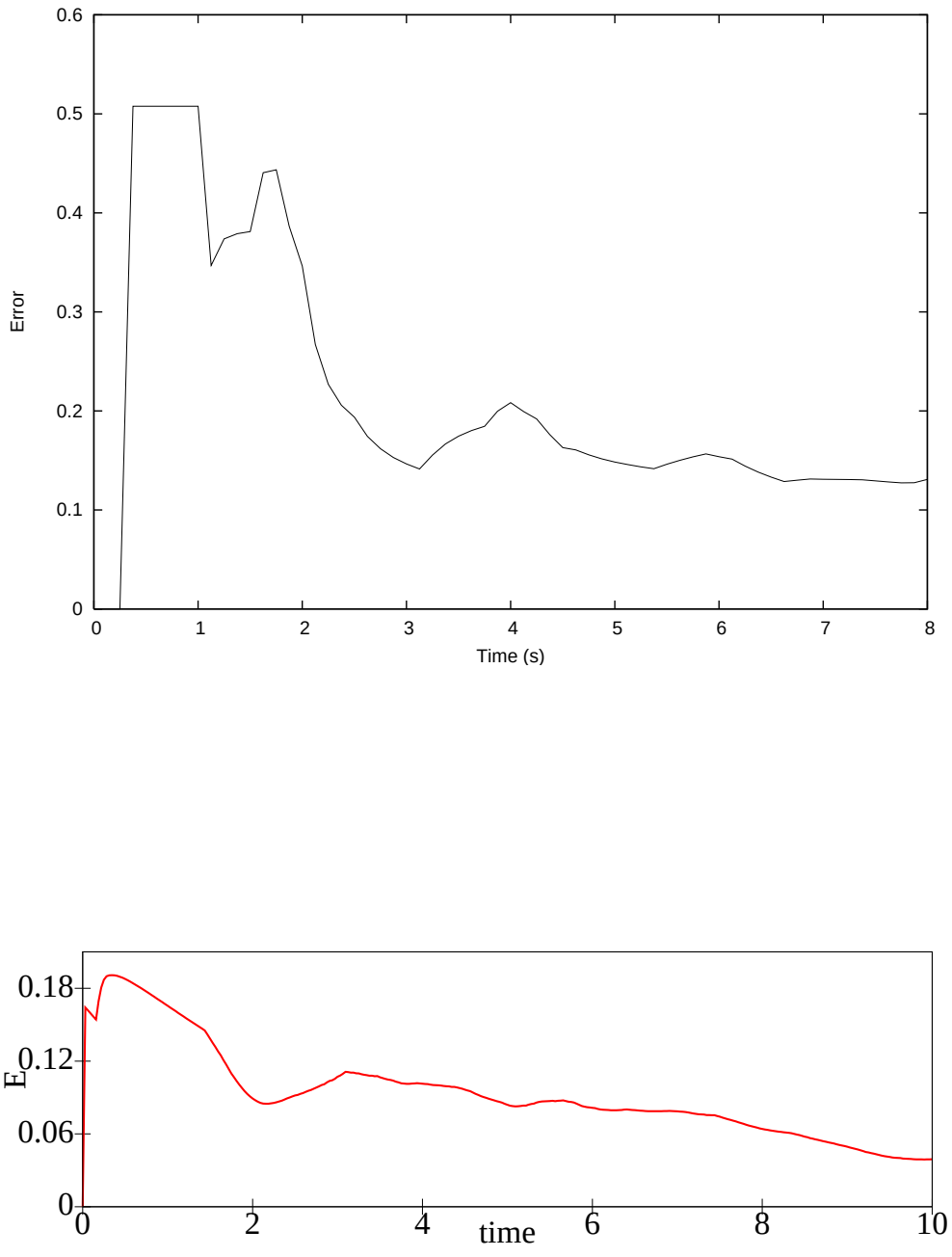


Figure 8.4: Tracking results from experiments (top) and simulation (bottom).

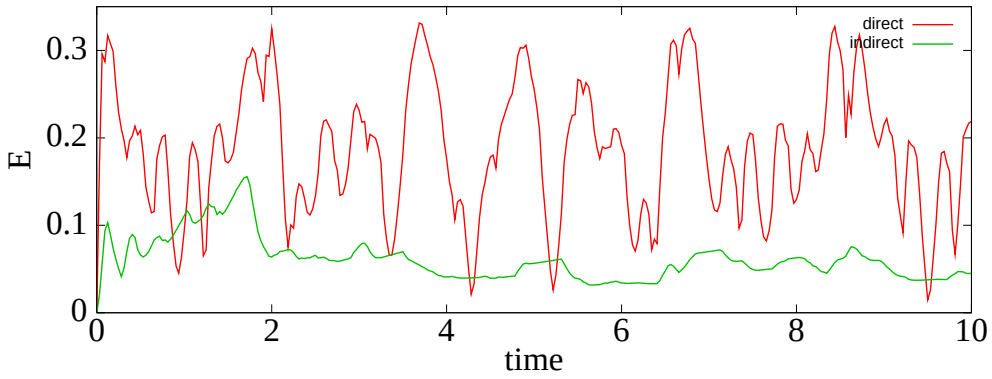


Figure 8.5: Performance of centroid methods.

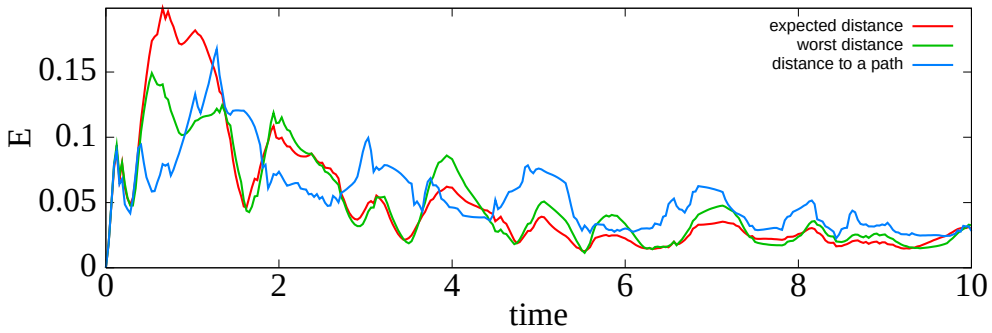


Figure 8.6: Position estimation errors of different weighting schemes.

Chapter 9

Structural Health Monitoring Application

Structural health monitoring can be an important tool for the ongoing maintenance of civil infrastructure. Given the size and complexity of many civil structures, a large network of sensors is required to adequately assess the structural condition. Traditional structural monitoring systems have been moving in the direction of dense deployment in recent years; however, the cost of installation can be thousands of dollars per sensor channel [13], and the amount of data generated by such a system can render the problem intractable [57]. Networks of wireless smart sensors have the potential to improve SHM dramatically by allowing for dense networks of sensors to be installed on a structure [24].

Acquiring pertinent information from these networks is accomplished by leveraging the on-board processing capability of the sensor nodes to implement distributed damage detection algorithms. One of the major roadblocks to achieving such a system is the magnitude and complexity of the required software development. The service-oriented architecture described in Chapter 3 can significantly simplify this process by offering a modular, reusable and extensible approach to software development.

Though the use of smart sensors in SHM has been previously demonstrated [42, 53], several hurdles have limited their widespread use for structural monitoring. In general, critical issues inherent in wireless sensor networks, such as synchronized sensing and data loss [64], still need to be addressed. In addition, the numerical algorithms required for system identification and damage detection must be implemented on sensor nodes that have limited resources. The result is that many SHM tasks require complex programming, ranging from network functionality to algorithm implementation. Application software development is made even more difficult by the fact that many smart sensor platforms employ special-purpose operating systems without support for common programming environments.

In this chapter, we demonstrate how our SOA approach can be applied in the SHM context address these challenges, and describe a long-term deployment of the resulting software system on a full-scale civil structure [1].

[1] Parts of this work appeared in previous publications [38, 76, 77].

9.1 Service-Oriented Architecture for SHM

We tackle the complexity associated with WSN software development for structural health monitoring by applying the design principles of service-oriented architecture outlined in Chapter 3. As a part of the Illinois Structural Health Monitoring Project (ISHMP), the Illinois SHM Services Toolsuite has been developed [36], providing a suite of services implementing key SHM algorithms for modal analysis and damage detection, along with the middleware infrastructure necessary to provide high-quality sensor data and to transport it reliably across the sensor network. This software is open-source and available for public use. As these services are loosely coupled and dynamically composable, different SHM applications can be easily created. Because it can be augmented with services for other domains, the tool suite also provides a common, extensible platform for WSN software development.

The wireless sensor platform used in this research is the Imote2 [2], which is well-suited to the demands of SHM applications. Tailored to the specific constraints of wireless sensor networks, TinyOS is the operating system used with this platform. While TinyOS has been adopted by many sensor network platforms and has quite a large user community, it can be a daunting undertaking for engineers lacking such specific programming experience to develop code for their applications.

The Illinois SHM Services Toolsuite provides an open-source software library of customizable services for, and examples of, SHM applications utilizing WSNs. The library is implemented on the Imote2 platform and allows SHM of structures in a distributed manner. The Toolsuite was originally based on the implementation of the Stochastic Damage Locating Vector (SDLV) algorithm [7], which requires synchronized acceleration data from a network of sensors and uses the Natural Excitation Technique (NExT) [37] and the Eigensystem Realization Algorithm (ERA) [40] for system identification. Further expansion of the Toolsuite provides alternative algorithms for system identification and distributed modal analysis,as well as a wide range of development tools and utilities.

9.1.1 SHM Application

The Illinois SHM Services Toolsuite was originally developed to implement the Distributed Computing Strategy (DCS) [24]. Nagayama and Spencer [62] and Nagayama et al. [64] developed the necessary middleware services, compiled the numerical sub-functions, and created the complex application code to implement DCS on a network of Imote2 sensors.

The sensor network is divided into overlapping clusters of sensors (Figure 9.1), which monitor sections of the structure. One of the nodes in each cluster, or local sensor community, is assigned as the *cluster head* and handles most of communication and data processing within the community. In addition to tasks inside

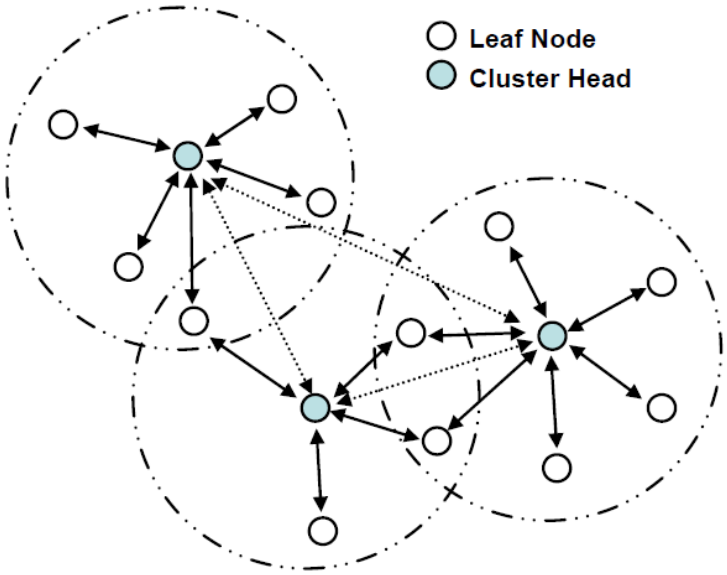


Figure 9.1: SHM application network topology.

the community, the cluster head communicates with the cluster heads of neighboring communities. When signaled to do so, all nodes in the network simultaneously measure acceleration and synchronize the data.

Upon the completion of synchronized sensing, the following tasks occur with in each local sensor community:

1. Acceleration data is shared to perform model-based data aggregation using NExT [64].

2. System identification is performed to determine the dynamic characteristics of the sub-component of the structure using ERA.

3. The SDLV algorithm is applied to determine if the structure has sustained any damage and to determine the location of the damage.

Finally, the cluster head of each local sensor community compares its results with the cluster heads of adjacent (and overlapping) communities to ensure that the results are consistent. If consensus between the cluster heads is achieved, only the outcome of the damage detection method is forwarded to the gateway node. This entire process is illustrated in the simplified flowchart shown in Figure 9.2.

This implementation has been experimentally verified on a laboratory scale three-dimensional truss. While the application proved to be successful in both network functionality and the ability to detect damage, it is not written in a modular way, thus lacking the flexibility necessary for adaptation to different applications. The following section describes the SOA that has been created from these middleware services and application code to provide the necessary modularity and flexibility.

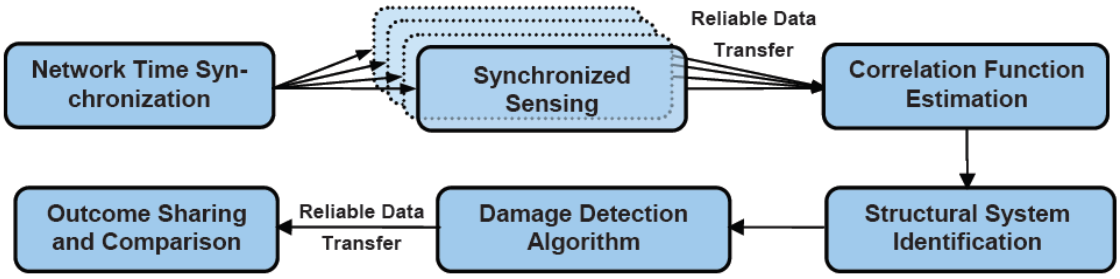


Figure 9.2: Flowchart of DCS implemented on a network of Imote2's.

While the Toolsuite was developed to implement DCS using the SDLV method on a network of Imote2's, its service-oriented architecture lends itself to further expansion and in the development of WSN applications for SHM. As a simple example, the figure below represents how the system identification method can be swapped out in an SHM application. In keeping with the SOA framework, these interchangeable services share the same input and output parameters. Other application examples that can benefit from the modular services provided in the toolsuite include distributed damage detection algorithms that rely only on the parameters derived from the correlation function estimates [12] or methods for distributed modal parameter estimation in a WSN [8].

9.1.2 SOA Implementation

The components of the service-based framework provided by the Illinois SHM Toolsuite can be divided into three primary categories: (1) foundation services, (2) application services and (3) tools and utilities. In addition, a library of supporting numerical functions that are common to many SHM algorithms is provided including fast Fourier transform (FFT), singular value decomposition (SVD), eigenvalue analysis, etc. In the description of the services that follows, *leaf nodes* are defined as the nodes that comprise the sensor network while the *gateway node* is the Imote2 that is connected to the base station PC that operates the network. The network topology that is utilized varies from application to application. Detailed descriptions of these applications and their network topologies can be found in [82, 77].

Foundation Services

In SOA terminology, services are high level, self-describing building blocks for distributed computing applications. The foundation services implement functionality needed to support the application and other services, rather than implementing application-specific tasks. They include gathering synchronized sensor data, reliably communicating both commands and long data records, and providing accurate and precise timestamps

to collected data. In combination, these services can be used by applications to achieve synchronized sensing from a network of sensors.

Unified Sensing: This service provides a convenient, general-purpose application programming interface (API) on top of the standard TinyOS sensing interface for the Imote2 platform, extending its functionality to include precise sample timestamping and providing transparent support for a variety of sensor boards. Data for all sensor channels, together with a single set of associated timestamps, is returned to the application in a single, shared data structure. A compact data representation format is used, which encapsulates all information necessary to recreate the sensor values, yet is memory-efficient for storage and transportation across the wireless network. This common data format is appropriate for passing directly to the application services described below. This complete and self-contained data is easy to pass around and modify without hard-coding connections between components that use only parts of this data [76].

Synchronized Virtual Sensing: The service described in Chapter 6 provides consistent, network-wide global timestamps for sensor data, making it possible to meaningfully compare data collected from multiple sensors. The resampling in this service is accomplished in a memory-efficient way with a resampling filter that is applied to the data one block of at a time [64], so that additional memory requirements for the service are independent of the size of the input data.

ReliableComm: Since sensor data loss is intrinsic to wireless systems and undermines the ability to perform system identification and detect damage [64], a reliable communication service, which eliminates data loss, is needed for sending commands and data between sensor nodes. Collectively termed "ReliableComm," this service combines four distinct reliable communication protocols, chosen automatically based on the type of communication, to eliminate data loss in an efficient manner. It also implements efficient data compression based on a custom Imote2 port of the zlib compression library [95], when data sizes are large enough to offset the computational overhead of compression and decompression.

Multi-hop Communication: This service implements a highly modified version of the AODV protocol [72]. It is tailored to data-intensive, bursty communication workloads common in SHM applications, and integrates transparently with ReliableComm to provide reliable multi-hop communication. A novel asymmetric routing metric that takes link quality into account favors reliable links, greatly speeding up reliable data transfer [63]. Multi-hop communication greatly expands the possible scale of SHM deployments utilizing the Toolsuite.

RemoteCommand: Provides an efficient means for the nodes to interact with each other to collaboratively conduct a certain task without the need to implement network communication protocols manually. RemoteCommand provides a fault tolerant and efficient implementation of Remote Method Invocation (RMI) [88] for the ISHMP Services Toolsuite. The most common pattern in WSN applications is that a node sends a command message to other nodes that in turn perform a task (e.g., time synchronization, sensing, or data processing) and optionally returns results to the original node. RemoteCommand, well-suited to the process, can significantly reduce the programming effort by taking advantage of the unique features offered by this service.

Application Services

These services provide the numerical algorithms necessary to implement SHM applications on the Imote2 and may also be used independently. For each application service, an application module to test the algorithm on both the PC and the Imote2 is provided. The numerical services are as follows:

CFE: Returns the Correlation Function Estimate (CFE) via FFT calculation. CFE uses two synchronized discrete-time signal vectors to obtain their CFE with user-specified number of FFT points and spectral window. In the case of DCS, the output of CFE is used as the input to the ERA or SSI system identification services.

ERA: Performs the Eigensystem Realization Algorithm (ERA). This time-domain system identification service uses the impulse-response function, or in the case of the NExT algorithm [37], the correlation functions, to determine the modal characteristics of the structure (damped natural frequencies, damping ratios, mode shapes, modal participation factors, EMAC values and the matrices which define the state-space model of the structure: A, B, and C).

SSI: Performs the covariance-driven Stochastic Subspace Identification (SSI) algorithm. This time-domain system identification method uses the cross correlation functions to determine the modal characteristics of the structure (damped natural frequencies, damping ratios, mode shapes, and the matrices which define the state-space model of the structure: A and C). Depending on the weighting function, the SSI is classified as (a) Balanced Realization (BR): no weighting, and (b) Canonical Variate Analysis (CVA): natural modes are balanced in terms of energy.

FDD: Performs the Frequency Domain Decomposition (FDD) algorithm [8]. This frequency-domain system identification method uses the cross spectra to determine the modal characteristics of the structure

(damped natural frequencies and mode shapes). Because the natural frequencies are selected by a peak-picking method, some modes may not be reliably found.

SDLV: Performs output-only, model-based damage detection using the Stochastic Damage Locating Vector (SDLV) method [62]. The inputs of SDLV are the modal characteristics determined by one of the system identification service.

Additional services are being developed by a community of civil engineers using the Toolsuite. Detailed on-line documentation is provided for each service and test application, giving more detail on requirements and formats of the inputs and outputs for the service [36].

Tools and Utilities

The application tools are complete applications that facilitate common tasks throughout the design, testing, deployment, and monitoring of the SHM applications, while utilities offer a set of basic testing and debugging commands to be included with existing applications. Included are utilities for resetting nodes remotely, listing the nodes within communication range of the local node, and changing the radio channel and power for local and remote nodes, and many others.

The application tools can be categorized as either those operating on a single node or those that distributed in the network. The single node application tools include:

autocomm: A basic terminal program for interfacing with the Imote2 through the Imote2 Interface Board's USB port. It uses the serial port UART interface to open a telnet-like connection with the mote. Features unique to this tool, which are not found in generic terminal applications, include: synchronization with the PC or other Imote2 gateway nodes over the Internet, periodic log file cycling, and e-mail notification of critical Imote2 errors.

TestServices: An example program that combines application services: CFE, ERA, and SDLV. It uses acceleration signals as input in the CFE service to calculate the correlation functions that is used in the ERA service. The estimated modal characteristics of the structure are then used in the SDLV service to identify damage.

The application tools that involve multiple nodes are as follows:

TestRadio: Tests the bidirectional communication quality between a sender node and a group of receiving nodes, and output the packet loss rate (in each direction, and round-trip). This is particularly for SHM applications, since the sensors are typically deployed close to complex civil structures, rendering communication quality unpredictable in advance.

RemoteSensing: A network-wide distributed application, this tool is used to collect sensor data from multiple sensors. Depending on the command that is given at run time, this services will either provide data that is nominally synchronized but not resampled or data that is resampled on prior to being centrally collected. In either case, the network is synchronized prior to sensing, then timestamped data is collected. If the resampling option is selected, the data is resampled locally using the SyncSensing service to account for any jitter or non-uniform delay in the start of sensing for each node. Sensor data is stored in flash on the leaf nodes, and can be retrieved immediately or at a later date. All data and commands in RemoteSensing are sent between nodes using the RemoteCommand service.

DecentralizedDataAggregation: This application is for data acquisition and processing based on decentralized hierarchical sensor network [82]. It supports multiple sensor clusters, in which data processing is conducted independently to other clusters. The main outputs of the application are sensor data and their correlation functions in each sensor cluster. Figure 9.3 shows a flowchart of this application.

The RemoteSensing and DecentralizedDataAggregation application tools employ the use of a distributed state machine to control the timing and flow of the application across a network of sensors. Internally, this is implemented via a shared SensingUnit component, which comes in centralized (RemoteSensing) and decentralized (DDA) variants. Figures 9.4 and 9.5 illustrate how the state machines operate for the RemoteSensing and DecentralizedDataAggregation applications, respectively. During each phase of the application, the sensor node is in a particular state that dictates its response to events and commands. Many additional applications have been developed on this base, and more are in development.

9.1.3 Example: Building an Application

The Illinois SHM Services Toolsuite facilitates a development environment in which services are easily integrated as building blocks to meet users need. DecentralizedDataAggregation, an application for modal analysis in decentralized WSN, is developed using the services. The DecentralizedDataAggregation application consists of two main parts: (a) sensing and (b) data processing (estimation of correlation function). For the sensing part, the Time Synchronization, the Unified Sensing, and the Synchronized Virtual Sensing services are combined to measure synchronized data (as in RemoteSensing), and in the data processing part, the CFE service utilizes the sensor data to calculate the correlation functions in each sensor clusters. The sensor data and the correlation functions are sent to the base station using the Reliable Communication service.

Upon designing an application for WSN, network topologies should be carefully considered. Because a centralized data acquisition system is not feasible for densely deployed smart sensors, a decentralized

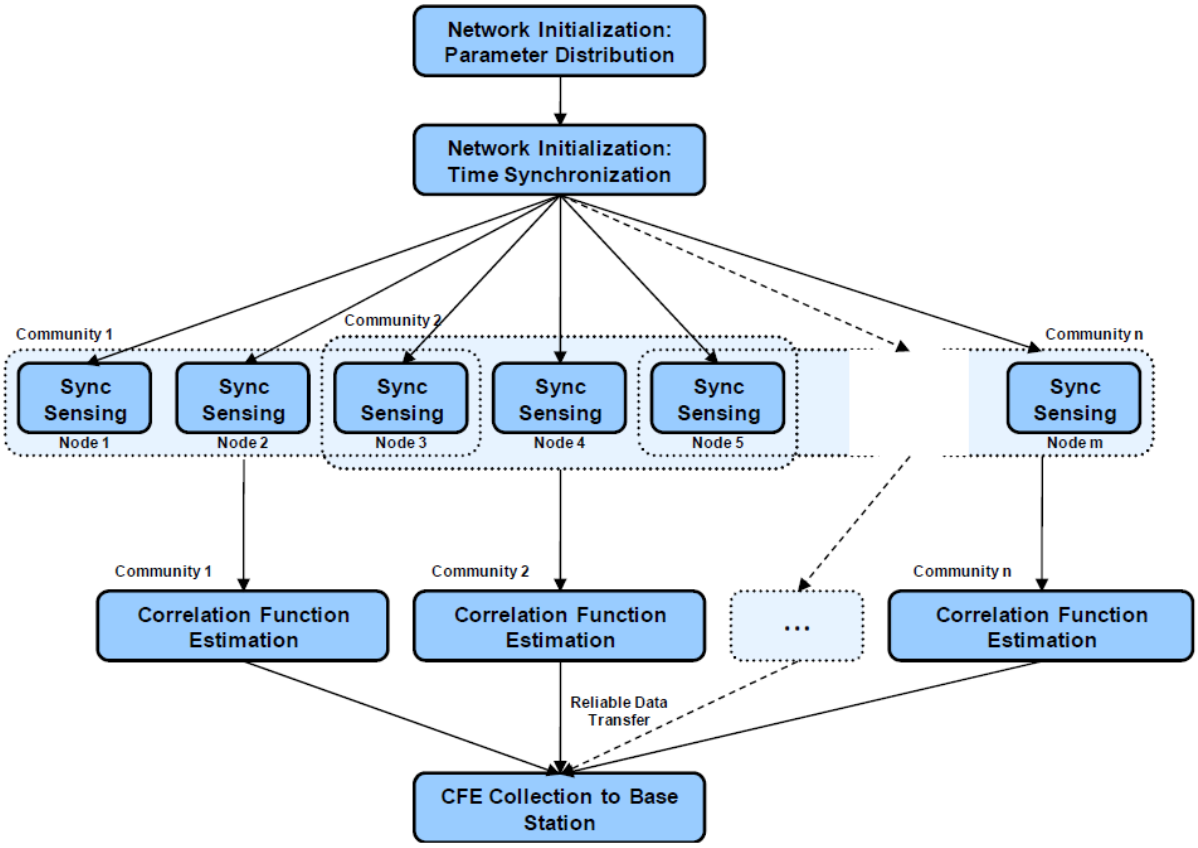


Figure 9.3: Information flowchart of DecentralizedDataAggregation.

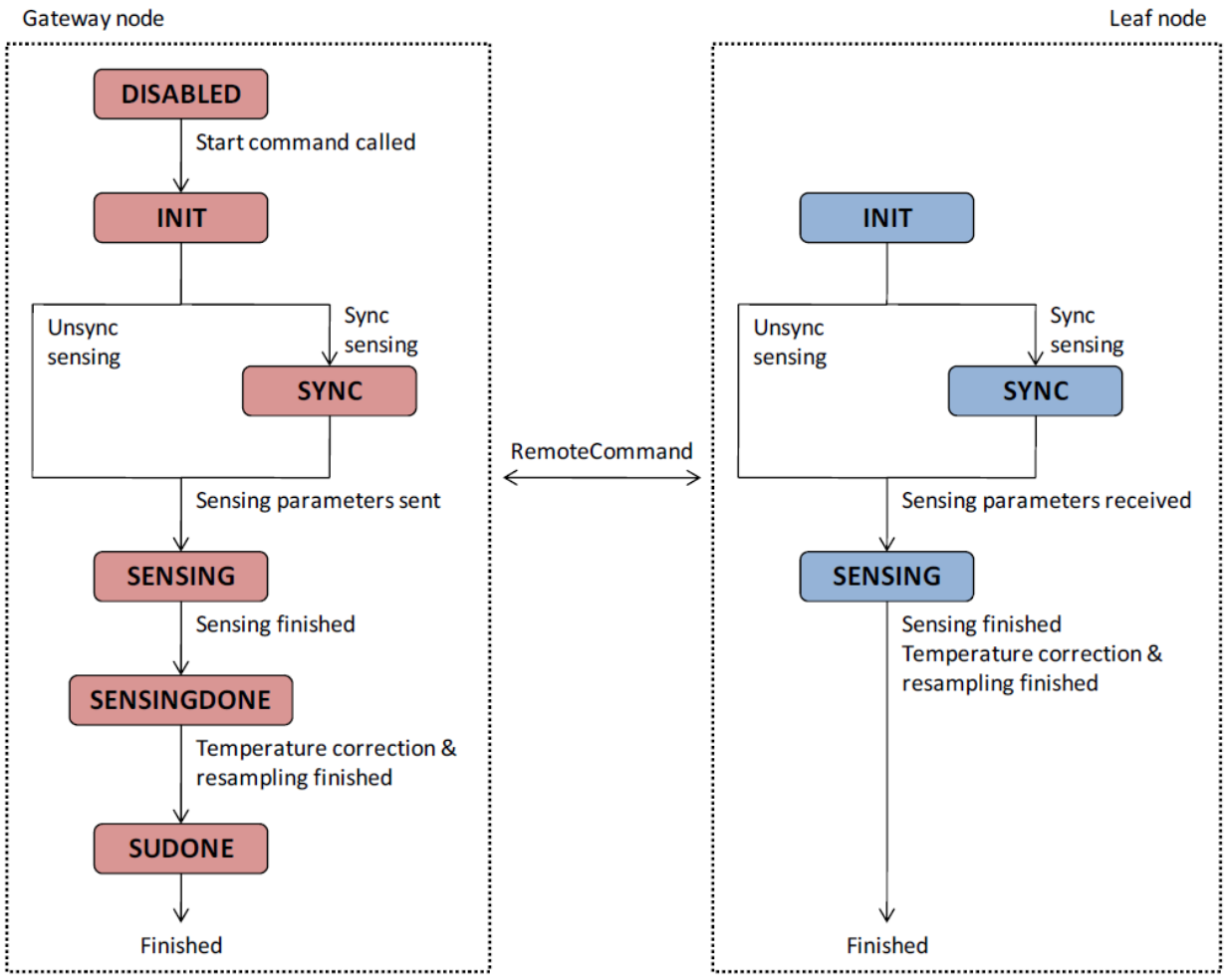


Figure 9.4: RemoteSensing state machine for the gateway (left) and leaf nodes (right).

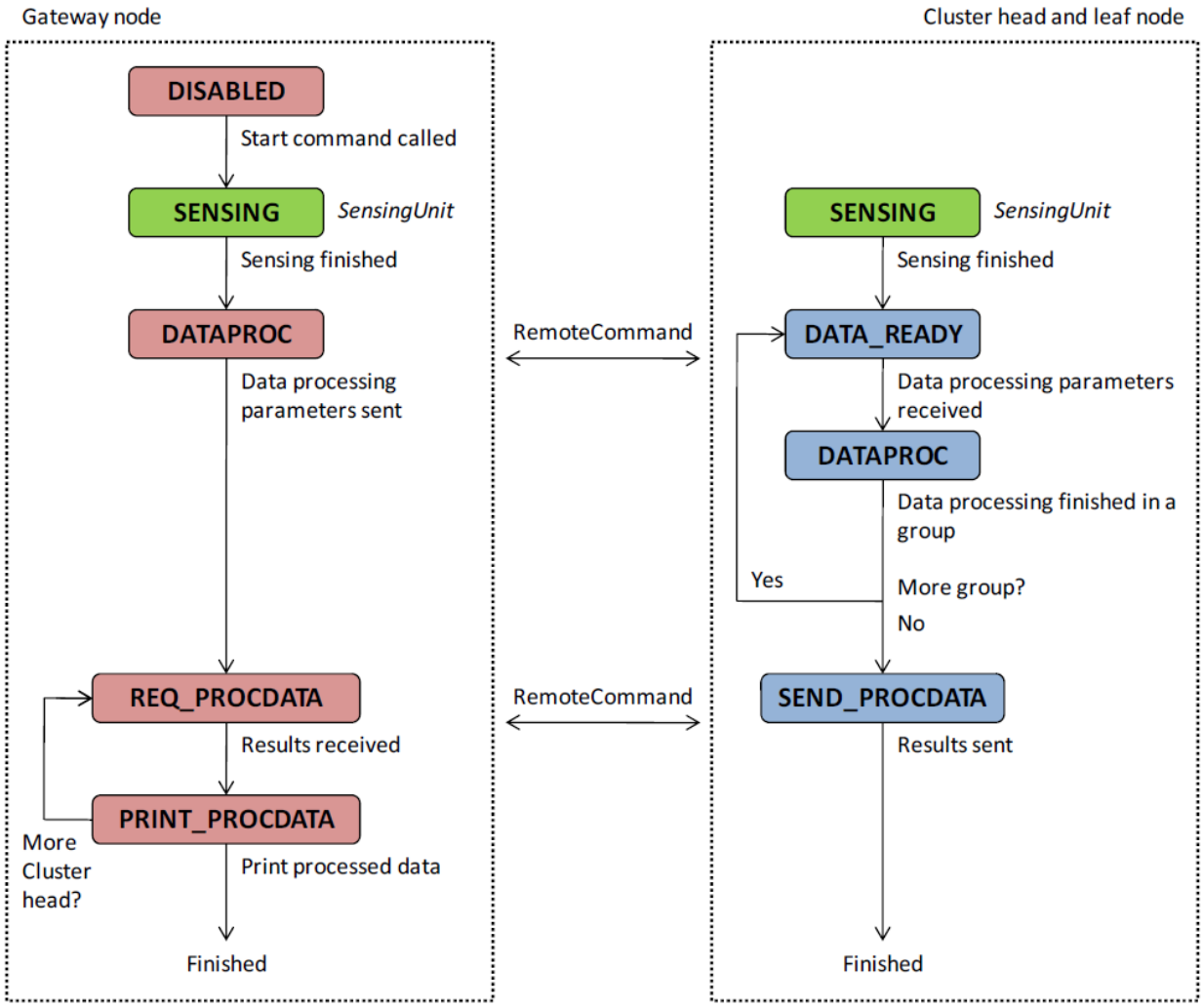


Figure 9.5: DDA state machine for the gateway (left) and the cluster head (right).

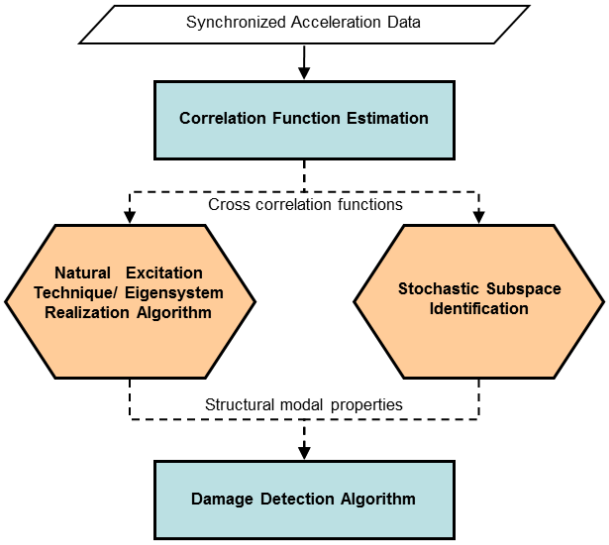


Figure 9.6: Alternative SHM approaches in a service-oriented application.

sensor network is preferred, in which data communication and processing occur in each sensor cluster independently [82]. Because many modal analysis methods require different types of inputs, the network topology should be determined according to the data processing method. DecentralizedDataAggregation is designed to calculate the correlation function with a single reference that can be used in ERA. For FDD and SSI/CVA to be embedded, the data processing part of the application should be designed to utilize multiple reference sensors. By considering the input for the modal analysis method in terms of the network topology, DecentralizedDataAggregation can be extended to include the services for modal analysis (ERA, SSI, and FDD) and damage detection (SDLV).

In addition to its inherent usefulness in decentralized SHM, DecentralizedDataAggregation provides a clear example to the users of the Illinois SHM Toolsuite of how to assemble the services into a full application by implementing the necessary control logic. Different SHM approaches can be implemented simply by replacing the relevant service in the application, as in Figure 9.6. In particular, this example illustrates the coordination of a distributed application by showing how an application's input parameters are specified by the user (at compilation and/or at run time), how each mote is functionally differentiated, how data is passed between services and how the scheduling of tasks takes place.

9.2 Full-Scale Experimental Validation

The software and hardware developed in this research was validated on a cable-stayed bridge in South Korea. The Jindo Bridge (Figure 9.7) connects Jindo Island to the Korean Peninsula near Haenam. This deployment is part of an international collaboration between South Korea (KAIST), Japan (University of Tokyo) and the USA (University of Illinois at Urbana-Champaign).

The primary goal of the Jindo Bridge deployment is to realize the first large-scale, autonomous network of smart sensors utilized for long-term SHM. The deployment has highlight the challenges and opportunities associated with such a large scale test-bed and thus has provided rich information for researchers and engineers interested in realizing a similar SHM system. For the research presented in this chapter, the primary goals of the deployment is to validate the capability of the SOA-based Illinois SHM Services Toolsuite to develop large-scale software for full-scale testing and autonomous network operation. In total, 113 Imote2 leaf nodes have been installed on the Jindo Bridge (Figure 9.8). In addition to ISM400 sensorboards used primarily for measuring acceleration, several high-sensitivity, strain- and wind-monitoring sensorboards were deployed.

The system has successfully captured ambient traffic loading with peak acceleration ranging from less than 5 mg to over 30 mg. Further analysis of the data resulted in the successful identification of the first twelve modes of vibration on the deck, as well as tension forces of 10 cables with large tensile stresses. The Jindo Bridge deployment has been documented in detail, including the implementation of renewable power sources, the selection and implication of various network parameters, and the analysis result of measured data [14, 38].

At the time of this writing, the sensor network remains in operation on the bridge.

Three critical requirements determine the success of a long-term monitoring WSN presented in this section: 1) continuous and autonomous monitoring, 2) efficient power management and 3) data inundation mitigation. While these requirements are often at odds, careful application design can mitigate this conflict. The solution is to implement a network that is only minimally active during non-critical structural response, but becomes fully active to measure higher response levels. The software presented in the previous section lays the groundwork for full-scale, autonomous monitoring of civil infrastructure.

Ideally, full-scale WSN deployments should require minimal external interaction after deployment (e.g., establishment of network operation parameters), unless instructed otherwise by the network administrator. Special care must be taken in the design of the application software to ensure a continuous and autonomous operating scenario is achieved while maintaining energy efficiency.

Figure 9.7: Twin Jindo Bridges connecting Jindo Island with the Korean Peninsula.

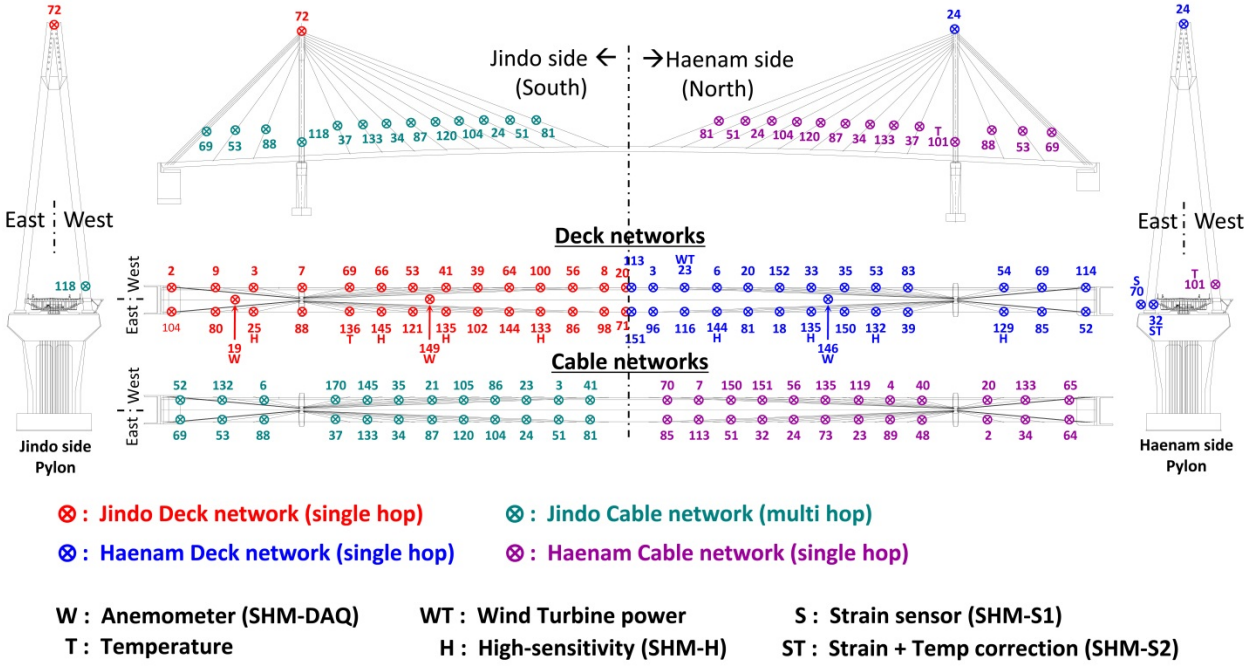


Figure 9.8: Deployment of 113 smart sensors for SHM of the Jindo Bridge.

9.2.1 Energy Management

In a traditional wired sensor implementations, energy management is of little concern. The sensors can remain active at all times and thus have the ability to be interrogated at any time to acquire data. Unlike wired systems, one of the most critical features of a successful WSN deployment is the implementation of careful energy management strategies. A common approach to achieving significant energy savings in sensor network applications is to allow the sensor nodes to sleep during periods of inactivity while waking periodically to listen for instructions [89]. The Imote2 allows the processor to be put into a deep sleep mode, whereby only the clock component of the processor is supplied power; all other components are powered down. Figure 9.9 illustrates the power savings associated with the deep sleep mode relative to other operations of the sensor node. When the sensor is in the deep sleep state, it cannot send data or receive commands via the radio or the serial port. Effectively, the node is inaccessible until the sleep time expires.

While it may seem advantageous to keep the leaf nodes in the deep sleep mode for extended periods of time to save power, this approach limits the ability of the gateway node to access the network at random to send inquiries or initiate network operations. To take advantage of the power savings of the deep sleep mode, while still allowing the gateway node access to the leaf nodes, a sleep/wake cycle service called SnoozeAlarm has been implemented in the ISHMP Toolsuite (Figure 9.10). When SnoozeAlarm operates on the leaf nodes (i.e., they are in the SnoozeAlarm mode), they sleep for a set period of time and then wake up for a relatively short period of time, during which they can listen and receive message. The ratio between the time spent awake and the sum of sleep time and the awake time is the SnoozeAlarm duty cycle. For optimal power saving, the duty cycle should be minimized while still allowing a long enough listen time to receive and process commands (approximately 500 ms). The actual overall power savings associated with the use of SnoozeAlarm is dependent on the application in which it is implemented.

The SnoozeAlarm wakeup command provides an efficient method for waking a network of leaf nodes in SnoozeAlarm mode. The ReliableComm protocol incorporates destination node wake-up from the SnoozeAlarm mode. When a leaf node sends back an acknowledgment, it is added to the list of nodes that have been successfully woken. This process continues until all nodes are awake, which in the worst case can take the full SnoozeAlarm duty cycle period.

In addition to energy conservation via deep sleep, the system uses rechargeable batteries with solar panels for all nodes (Figure 9.11). Since the charging current supplied by the solar panels varies unpredictably from node to node and with weather conditions and time of day, a one-size-fits-all power management plan is untenable (Figure 9.12). A dynamic energy monitoring service is employed to locally adjust the behavior of each node based on its remaining battery power level and charging conditions. If battery power is critically

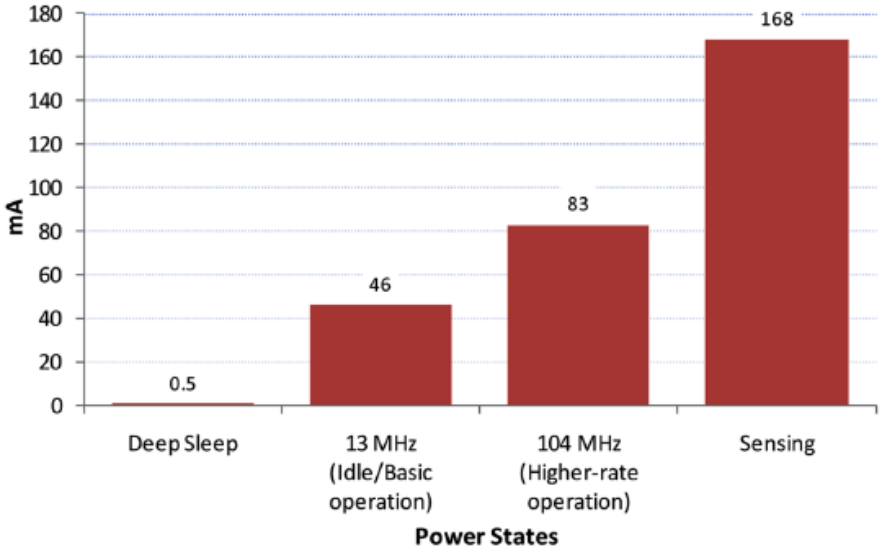


Figure 9.9: Current draw in various operational modes of the Imote2 with the ISM400 sensor board.

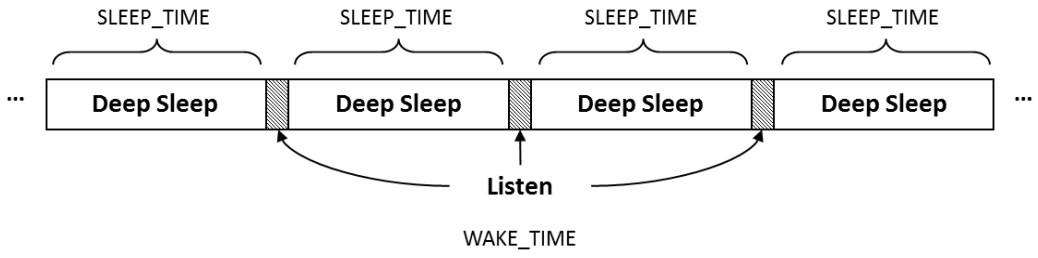


Figure 9.10: SnoozeAlarm periodic wake-up cycle.

Figure 9.11: Solar-powered sensor node attached to a bridge cable.

low and the node is in danger of failing, it stops participating in normal network operations and is put into deep sleep for an extended period. The lifetime of the network is thus prioritized over the temporary loss of data from individual nodes.

9.2.2 Sentry-triggered Data Acquisition

One approach to ensure that important occurrences are captured by the sensor network is to designate a subset of the network to sense data on a more frequent basis than the rest of the network and provide alerts [34, 89]. In this research, the ThresholdSentry service allows a subset of the leaf nodes to act as sentry nodes, in addition to their duties as leaf nodes. The selection of sentry nodes and their threshold values should be made such that the threshold is exceeded often enough for adequate structural monitoring. ThresholdSentry nodes are statically chosen and are equipped with a dual solar cell to account for increased energy consumption.

If a sentry node determines that a configurable threshold value has been exceeded during its observation time, it notifies the gateway node and initiates network-wide sensing. The current implementation of ThresholdSentry can utilize either acceleration or wind speed measurements; however, other triggers, such as strain levels , may be incorporated into the application.

The current implementation of ThresholdSentry used in conjunction with RemoteSensing allows the network to capture the occurrence of longer-duration, lower-frequency events such as high wind; however, it does not support capturing short-term, transient events such as an earthquake. The time required to wake

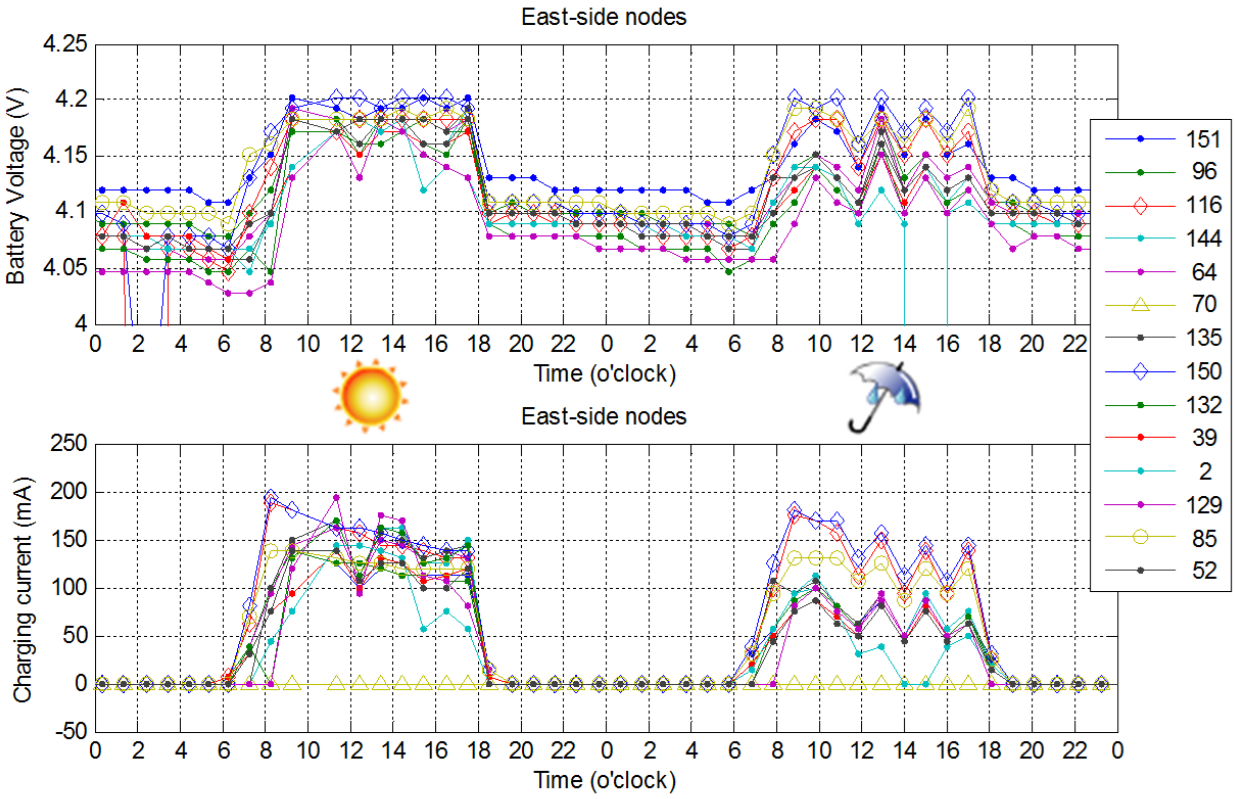


Figure 9.12: Current supplied by the solar panel to the rechargeable battery.

the network and perform time synchronization prior to the collection of data would cause such events to be missed. In future research, the network wakeup time could be reduced using a propagating wakeup message with optimized communication parameters and coordinating sleep cycles among the sensor nodes.

9.2.3 Autonomous Long-term Monitoring

Achieving an autonomous SHM implementation on a network of smart sensors requires a high-level application to coordinate its components in response to various events and schedule global network activities. *AutoMonitor* has been developed to provide this functionality. AutoMonitor operates on the gateway node and serves the purpose of maintaining the RemoteSensing and ThresholdSentry parameters and coordinating the associated network tasks.

AutoMonitor is programmed by means of an input file created by the user of the network. Once started, it requires no additional input from the user. Consisting of a simple language for batch execution of parametrized commands, virtually no programming knowledge is required to program AutoMonitor's behavior. In fact, a domain expert is expected to have a better knowledge of the relevant parameters than an embedded systems programmer. The selection of most of these parameters is highly application-dependent and will take a period of adjustment and refinement to optimize for each case. Many of these parameters have power consumption implications; their effect on power management must be carefully considered [38] describes the selection of these parameters for the long span bridge deployment described in the following section.

After the input file containing all of the necessary parameters is read in, AutoMonitor initiates scheduled periodic monitoring and execution of applications such as RemoteSensing, ThresholdSentry, and others. AutoMonitor employs a timer and a counter to limit the number of monitoring events that occur within a particular time period to limit burst power consumption.

9.3 Discussion

This chapter describes an open-source services toolsuite that was developed using the design principles of service-oriented architecture. This SOA-based approach creates a framework which allows application programmers to more easily create applications for SHM systems deployed on WSNs. As a result, the framework allows more researchers, and ultimately application engineers, to design and implement successful SHM systems without the requirement of how the underlying middleware and application services are implemented. In its current form, toolsuite requires application programmers to provide only the code necessary to interconnect the services and tools in a way that makes sense for their applications.

A state-of-the-art SHM system using a WSN has been successfully deployed on the Jindo Bridge to verify the performance of the system and to serve as a driver for advancement of smart sensor technology. An autonomous structural monitoring system based on the ISHMP Toolsuite software, employs a threshold detection strategy and an energy-efficient sleeping mode to extend the network lifetime indefinitely, in conjunction with solar-powered nodes providing energy harvesting for powering the sensor network.

In total, 113 sensor nodes have been installed, divided into four sub-networks to decrease the communication time and increase network throughput. All sensors are carefully located based on radio communication capability determined by extensive radio communication tests. The measured data shows a good agreement with data from the existing wired system, which verifies that the data quality of the WSN is reliable. Successful deployment of this WSN demonstrates the suitability of the Illinois SHM Toolsuite as a framework for the creation of large-scale sensor network systems for long-term, continuous, autonomous SHM.

The software components presented in this chapter enable continuous operation of WSNs for SHM applications outside of the laboratory setting. This software allows critical structural response to be captured while maintaining low-power network operation the majority of the time. The AutoMonitor network management application coordinates the operation of ThresholdSentry, SnoozeAlarm and RemoteSensing to ensure autonomous and continuous functionality of the network.

Chapter 10

Conclusions

Our research creates a unique dynamic macroprogramming platform for sensor networks. The results of this thesis demonstrate that programmability of complex cyber-physical systems may be dramatically improved by separating context-independent application logic, known at design time, from the low-level execution context considerations that are often unavailable until run time. The Ambiance dynamic macroprogramming platform is able to accomplish this goal. The approach taken by Ambiance is distinct from compilation-based systems such as Regiment and ATaG in that it enables dynamic service composition and adaptation to changes in the execution environment. The use of Dart and the AOM architectural style to represent macroprograms exclusively in domain-specific terms separates it from existing macroprogramming systems based on fixed languages or abstractions.

A concrete contribution of this research is the implementation of a service-oriented software development toolsuite for the structural health monitoring application domain. The ease of application development compared to the state of the art is expected to attract new interest to WSN-based solutions in that field, and facilitate research and development of new damage detection algorithms, tools, and middleware services for SHM. Indeed, over 70 research groups worldwide have adopted the Illinois SHM software.

We believe that the design principles and architecture set forth in this work have wider implications beyond the adaptive execution of composite service-oriented workloads in WSNs. The dynamic macroprogramming approach favors a scalable, robust application development process, where the application designer does not have to explicitly specify all aspects of the low-level execution of the system. However, an important aspect of the system relegated to future work is a study of which aspects of low-level service optimization and control decisions can be externalized to the macroprogramming layer. We consider system-wide optimization of independent concurrent applications in a shared WSN environment to be a major open research topic.

Ambient Intelligence technologies, once mature, will enable novel tools and new work practices in many fields. Ambient computing infrastructure will provide for the wide availability of real-time sensor data, enabling informed decision making and rapid response. We believe dynamic macroprogramming for WSNs, as embodied by the Ambiance platform, is a necessary step toward enabling true Ambient Intelligence.

References

[1] H. Abelson, G. Sussman, and J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd ed. MIT Press, 1996. [Online]. Available: http://mitpress.mit.edu/sicp/

[2] R. Adler, M. Flanigan, J. Huang, R. Kling, N. Kushalnagar, L. Nachman, C.-Y. Wan, and M. Yarvis, "Intel Mote 2: an advanced platform for demanding sensor network applications," in *Proceedings of the 3rd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '05. New York, NY, USA: ACM, 2005, pp. 298–298. [Online]. Available: http://doi.acm.org/10.1145/1098918.1098963

[3] G. Agha, *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press, 1986.

[4] G. Agha, C. Houck, and R. Panwar, "Distributed execution of actor programs," in *Proceedings of the Fourth International Workshop on Languages and Compilers for Parallel Computing*. London, UK: Springer-Verlag, 1992, pp. 1–17. [Online]. Available: http://portal.acm.org/citation.cfm?id=645669.665212

[5] J. Aslam, Z. Butler, F. Constantin, V. Crespi, G. Cybenko, and D. Rus, "Tracking a moving object with a binary sensor network," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 150–161. [Online]. Available: http://doi.acm.org/10.1145/958491.958509

[6] J. Aspnes, D. Goldenberg, and Y. Yang, "On the computational complexity of sensor network localization," in *Algorithmic Aspects of Wireless Sensor Networks*, ser. Lecture Notes in Computer Science, S. Nikoletseas and J. Rolim, Eds. Springer, 2004, vol. 3121, pp. 32–44. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-27820-7_5

[7] D. Bernal, "Flexibility-based damage localization from stochastic realization results," *Engineering Mechanics*, vol. 132, no. 6, pp. 651–658, 2006. [Online]. Available: http://dx.doi.org/doi/10.1061/(ASCE)0733-9399(2006)132:6(651)

[8] R. Brincker, L. Zhang, and P. Anderson, "Modal identification of output-only systems using frequency domain decomposition," *Smart Materials and Structures*, vol. 10, no. 3, pp. 441–445, 2001. [Online]. Available: http://dx.doi.org/10.1088/0964-1726/10/3/303

[9] R. Brooks, C. Griffin, and D. Friedlander, "Self-organized distributed sensor network entity tracking," *High Performance Computing Applications*, vol. 16, no. 3, pp. 207–219, August 2002. [Online]. Available: http://dx.doi.org/10.1177/10943420020160030201

[10] Q. Cao, T. Abdelzaher, J. Stankovic, and T. He, "The LiteOS operating system: Towards Unix-like abstractions for wireless sensor networks," in *Proceedings of the 7th International Conference on Information Processing in Sensor Networks*, ser. IPSN '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 233–244. [Online]. Available: http://dx.doi.org/10.1109/IPSN.2008.54

[11] N. Carriero and D. Gelernter, "Linda in context," *Communications of the ACM*, vol. 32, pp. 444–458, April 1989. [Online]. Available: http://doi.acm.org/10.1145/63334.63337

[12] N. Castaneda, F. Sun, S. Dyke, C. Lu, A. Hope, and T. Nagayama, "Implementation of a correlation-based decentralized damage detection method using wireless sensors," in *Proceedings of the International Conference on Advances in Structural Engineering and Mechanics*, May 2008.

[13] M. Celebi, "Seismic instrumentation of buildings (with emphasis on federal buildings)," United States Geological Survey, Tech. Rep. 0-7460-68170, 2002.

[14] S. Cho, S. A. Jang, H. Jo, K. Mechitov, J. Rice, H.-J. Jung, C.-B. Yun, B. F. Spencer, T. Nagayama, and J. Seo, "Structural health monitoring system of a cable-stayed bridge using a dense array of scalable smart sensor network," *Sensors and Smart Structures Technologies for Civil, Mechanical, and Aerospace Systems*, vol. 7647, no. 1, 2010. [Online]. Available: http://link.aip.org/link/?PSI/7647/764707/1

[15] T. F. Cox and M. A. A. Cox, *Multidimensional Scaling*, 2nd ed. Chapman and Hall, 2000.

[16] K. Delin, S. Jackson, D. Johnson, S. Burleigh, R. Woodrow, J. McAuley, J. Dohm, F. Ip, T. Ferr, D. Rucker, and V. Baker, "Environmental studies with the sensor web: Principles and practice," *Sensors*, vol. 5, pp. 103–117, 2005.

[17] S. Doebling, C. Farrar, M. Prime, and D. Shevitz, "Damage identification and health monitoring of structural and mechanical systems from changes in their vibration characteristics: a literature review," Los Alamos National Laboratory, Tech. Rep. LA–13070-MS, May 1996.

[18] A. Dunkels, B. Gronvall, and T. Voigt, "Contiki - a lightweight and flexible operating system for tiny networked sensors," in *Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, ser. LCN '04. Washington, DC, USA: IEEE Computer Society, 2004, pp. 455–462. [Online]. Available: http://dx.doi.org/10.1109/LCN.2004.38

[19] 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 *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, ser. IPSN '05. Piscataway, NJ, USA: IEEE Press, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1147685.1147772

[20] R. K. Dybvig, *The Scheme Programming Language*, 3rd ed. MIT Press, 2003. [Online]. Available: http://www.scheme.com/tspl3/

[21] J. Elson, L. Girod, and D. Estrin, "Fine-grained network time synchronization using reference broadcasts," *SIGOPS Operating Systems Review*, vol. 36, pp. 147–163, December 2002. [Online]. Available: http://doi.acm.org/10.1145/844128.844143

[22] R. Fletcher, *Practical Methods of Optimization*, 2nd ed. Wiley, 2000.

[23] C.-L. Fok, G.-C. Roman, and C. Lu, "Mobile agent middleware for sensor networks: an application case study," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, ser. IPSN '05. Piscataway, NJ, USA: IEEE Press, 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1147685.1147747

[24] Y. Gao and B. F. Spencer, "Structural health monitoring strategies for smart sensor networks," Newmark Structural Engineering Laboratory, University of Illinois at Urbana-Champaign, Tech. Rep. 011, May 2008. [Online]. Available: http://hdl.handle.net/2142/8802

[25] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler, "The nesC language: a holistic approach to networked embedded systems," in *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, ser. PLDI '03. New York, NY, USA: ACM, 2003, pp. 1–11. [Online]. Available: http://doi.acm.org/10.1145/781131.781133

[26] V. Ginot, C. Le Page, and S. Souissi, "A multi-agents architecture to enhance end-user individual-based modelling," *Ecological Modelling*, vol. 157, no. 1, pp. 23–41, 2002. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0304380002002119

[27] O. Gnawali, K.-Y. Jang, J. Paek, M. Vieira, R. Govindan, B. Greenstein, A. Joki, D. Estrin, and E. Kohler, "The Tenet architecture for tiered sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 153–166. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182823

[28] T. Gu, H. K. Pung, and D. Q. Zhang, "A service-oriented middleware for building context-aware services," *Network and Computer Applications*, vol. 28, pp. 1–18, January 2005. [Online]. Available: http://portal.acm.org/citation.cfm?id=1053030.1053031

[29] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava, "A dynamic operating system for sensor nodes," in *Proceedings of the 3rd International Conference on Mobile Systems, Applications, and Services*, ser. MobiSys '05. New York, NY, USA: ACM, 2005, pp. 163–176. [Online]. Available: http://doi.acm.org/10.1145/1067170.1067188

[30] C. E. Hewitt, "Viewing control structures as patterns of passing messages," *Artificial Intelligence*, vol. 8, no. 3, pp. 323–364, 1977.

[31] J. L. Hill and D. E. Culler, "Mica: a wireless platform for deeply embedded networks," *IEEE Micro*, vol. 22, no. 6, pp. 12–24, November/December 2002.

[32] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *SIGPLAN Notices*, vol. 35, pp. 93–104, November 2000. [Online]. Available: http://doi.acm.org/10.1145/356989.356998

[33] B. Hofmann-Wellenhof, H. Lichtenegger, and J. Collins, *Global Positioning System: Theory and Practice*, 5th ed. Springer-Verlag, 2001.

[34] J. Hui, Z. Ren, and B. H. Krogh, "Sentry-based power management in wireless sensor networks," in *Proceedings of the 2nd International Conference on Information Processing in Sensor Networks*, ser. IPSN'03. Springer-Verlag, 2003, pp. 458–472. [Online]. Available: http://portal.acm.org/citation.cfm?id=1765991.1766023

[35] J. W. Hui and D. Culler, "The dynamic behavior of a data dissemination protocol for network programming at scale," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 81–94. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031506

[36] Illinois Structural Health Monitoring Project, "Illinois SHM Services Toolsuite." [Online]. Available: http://shm.web.cs.illinois.edu/software.html

[37] G. James, T. Carne, and J. Lauffer, "The natural excitation technique for modal parameter extraction from operating wind turbine," Sandia National Laboratories, Tech. Rep. SAND92-1666 UC-261, 2003.

[38] S. Jang, H. Jo, S. Cho, K. Mechitov, J. Rice, S.-H. Sim, H.-J. Jung, C.-B. Yun, B. F. Spencer, and G. Agha, "Structural health monitoring of a cable-stayed bridge using smart sensor technology: deployment and evaluation," *Smart Structures and Systems*, vol. 6, no. 5, pp. 439–460, 2010. [Online]. Available: http://hdl.handle.net/2142/16434

[39] E. Johnson, P. Voulgaris, and L. Bergman, "Methods of system identification for monitoring slowly time-varying structural systems," in *Proceedings of the Intelligent Information Systems Conference*, December 1997, pp. 569–573. [Online]. Available: http://dx.doi.org/10.1109/IIS.1997.645422

[40] J.-N. Juang and R. S. Pappa, "An eigensystem realization algorithm for modal parameter identification and model reduction," *Guidance, Control, and Dynamics*, vol. 8, no. 5, pp. 620–627, 1985. [Online]. Available: http://dx.doi.org/10.2514/3.20031

[41] S. N. Kamin, *Programming Languages: An Interpreter-Based Approach*. Boston, MA, USA: Addison-Wesley, 1990.

[42] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon, "Health monitoring of civil infrastructures using wireless sensor networks," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, ser. IPSN '07. New York, NY, USA: ACM, 2007, pp. 254–263. [Online]. Available: http://doi.acm.org/10.1145/1236360.1236395

[43] W. Kim, K. Mechitov, J.-Y. Choi, and S. Ham, "On target tracking with binary proximity sensors," in *Proceedings of the 4th International Symposium on Information Processing in Sensor Networks*, ser. IPSN '05. Piscataway, NJ, USA: IEEE Press, 2005, pp. 301–308. [Online]. Available: http://portal.acm.org/citation.cfm?id=1147685.1147735

[44] Y. Kwon, K. Mechitov, S. Sundresh, W. Kim, and G. Agha, "Resilient localization for sensor networks in outdoor environments," *ACM Transactions on Sensor Networks*, vol. 7, no. 1, pp. 3:1–3:30, August 2010. [Online]. Available: http://doi.acm.org/10.1145/1806895.1806898

[45] Y. Kwon, S. Sundresh, K. Mechitov, and G. Agha, "ActorNet: an actor platform for wireless sensor networks," in *Proceedings of the 5th International Joint Conference on Autonomous Agents and Multiagent Systems*, ser. AAMAS '06. New York, NY, USA: ACM, 2006, pp. 1297–1300. [Online]. Available: http://doi.acm.org/10.1145/1160633.1160871

[46] R. Lajara, J. Pelegri-Sebastia, and J. J. P. Solano, "Power consumption analysis of operating systems for wireless sensor networks," *Sensors*, vol. 10, no. 6, pp. 5809–5826, 2010. [Online]. Available: http://www.mdpi.com/1424-8220/10/6/5809/

[47] L. Lamport and P. M. Melliar-Smith, "Synchronizing clocks in the presence of faults," *Journal of the ACM*, vol. 32, pp. 52–78, January 1985. [Online]. Available: http://doi.acm.org/10.1145/2455.2457

[48] P. Levis and D. Culler, "Maté: a tiny virtual machine for sensor networks," *SIGPLAN Notices*, vol. 37, pp. 85–95, October 2002. [Online]. Available: http://doi.acm.org/10.1145/605432.605407

[49] D. Li, K. Wong, Y. H. Hu, and A. Sayeed, "Detection, classification, and tracking of targets," *IEEE Signal Processing Magazine*, vol. 19, no. 2, pp. 17–29, March 2002. [Online]. Available: http://dx.doi.org/10.1109/79.985674

[50] H. B. Lim, Y. M. Teo, P. Mukherjee, V. T. Lam, W. F. Wong, and S. See, "Sensor grid: Integration of wireless sensor networks and the grid," in *Proceedings of the Annual IEEE Conference on Local Computer Networks*. Los Alamitos, CA, USA: IEEE Computer Society, 2005, pp. 91–99. [Online]. Available: http://doi.ieeecomputersociety.org/10.1109/LCN.2005.123

[51] J. Liu and F. Zhao, "Towards semantic services for sensor-rich information systems," in *Proceedings of the 2nd International Conference on Broadband Networks*, vol. 2, October 2005, pp. 967–974. [Online]. Available: http://dx.doi.org/10.1109/ICBN.2005.1589709

[52] L. Luo, T. F. Abdelzaher, T. He, and J. A. Stankovic, "EnviroSuite: an environmentally immersive programming framework for sensor networks," *Transactions on Embedded Computer Systems*, vol. 5, pp. 543–576, August 2006. [Online]. Available: http://doi.acm.org/10.1145/1165780.1165782

[53] J. Lynch and K. Loh, "A summary review of wireless sensors and sensor networks for structural health monitoring," *Shock and Vibration Digest*, vol. 38, no. 2, p. 91, 2006.

[54] G. Mainland, L. Kang, S. Lahaie, D. C. Parkes, and M. Welsh, "Using virtual markets to program global behavior in sensor networks," in *Proceedings of the 11th ACM SIGOPS European Workshop*, ser. EW 11. New York, NY, USA: ACM, 2004. [Online]. Available: http://doi.acm.org/10.1145/1133572.1133587

[55] A. Mainwaring, D. Culler, J. Polastre, R. Szewczyk, and J. Anderson, "Wireless sensor networks for habitat monitoring," in *Proceedings of the 1st ACM International Workshop on Wireless Sensor Networks and Applications*, ser. WSNA '02. New York, NY, USA: ACM, 2002, pp. 88–97. [Online]. Available: http://doi.acm.org/10.1145/570738.570751

[56] M. Maróti, B. Kusy, G. Simon, and A. Lédeczi, "The flooding time synchronization protocol," in *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, ser. SenSys '04. New York, NY, USA: ACM, 2004, pp. 39–49. [Online]. Available: http://doi.acm.org/10.1145/1031495.1031501

[57] K. Mechitov, W. Kim, G. Agha, and T. Nagayama, "High-frequency distributed sensing for structure monitoring," *Transactions of the Society of Instrument and Control Engineers*, vol. E-S-1, no. 1, pp. 109–114, 2006. [Online]. Available: http://osl.web.cs.illinois.edu/docs/sice05mechitov/sice06mechitov.pdf

[58] K. Mechitov, R. Razavi, and G. Agha, "Architecture design principles to support adaptive service orchestration in WSN applications," *SIGBED Review*, vol. 4, no. 3, pp. 37–42, July 2007. [Online]. Available: http://doi.acm.org/10.1145/1317103.1317110

[59] K. Mechitov, S. Sundresh, Y. Kwon, and G. Agha, "Cooperative tracking with binary-detection sensor networks," in *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, ser. SenSys '03. New York, NY, USA: ACM, 2003, pp. 332–333. [Online]. Available: http://doi.acm.org/10.1145/958491.958546

[60] K. Mechitov, S. Sundresh, Y. Kwon, and G. Agha, "Cooperative tracking with binary-detection sensor networks," Department of Computer Science, University of Illinois at Urbana-Champaign, Tech. Rep. UIUCDCS-R-2003-2379, 2003. [Online]. Available: http://osl.web.cs.illinois.edu/docs/mechitov03uiuc/tracking.pdf

[61] D. Mills, J. Martin, J. Burbank, and W. Kasch, "Network Time Protocol version 4: Protocol and algorithms specification," Internet Engineering Task Force, Tech. Rep. RFC 5905, June 2010. [Online]. Available: http://www.rfc-editor.org/info/rfc5905

[62] T. Nagayama and B. Spencer, "Structural health monitoring using smart sensors," Newmark Structural Engineering Laboratory, University of Illinois at Urbana-Champaign, Tech. Rep. 001, 2007. [Online]. Available: http://hdl.handle.net/2142/3521

[63] T. Nagayama, P. Moinzadeh, K. Mechitov, M. Ushita, N. Makihata, M. Ieiri, G. Agha, B. F. Spencer, Y. Fujino, and J.-W. Seo, "Reliable multi-hop communication for structural health monitoring," *Smart Structures and Systems*, vol. 6, no. 5, pp. 481–504, 2010. [Online]. Available: http://hdl.handle.net/2142/16434

[64] T. Nagayama, B. F. Spencer, K. Mechitov, and G. Agha, "Middleware services for structural health monitoring using smart sensors," *Smart Structures and Systems*, vol. 5, no. 2, 2008. [Online]. Available: http://osl.web.cs.illinois.edu/docs/sss08/sss08.pdf

[65] B. Nardi, *A Small Matter of Programming: Perspectives on End User Computing*. MIT Press, 1993.

[66] National Research Council, *Embedded Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. Washington, DC: National Academy Press, 2001.

[67] R. Newton, G. Morrisett, and M. Welsh, "The regiment macroprogramming system," in *Proceedings of the 6th International Conference on Information Processing in Sensor Networks*, ser. IPSN '07. New York, NY, USA: ACM, 2007, pp. 489–498. [Online]. Available: http://doi.acm.org/10.1145/1236360.1236422

[68] D. Niculescu and B. Nath, "Ad hoc positioning system (APS)," in *IEEE Global Telecommunications Conference*, ser. GLOBECOM '01, vol. 5, 2001, pp. 2926–2931. [Online]. Available: http://dx.doi.org/10.1109/GLOCOM.2001.965964

[69] A. Oppenheim, R. Schafer, and J. Buck, *Discrete-time signal processing*. Prentice Hall, 1999.

[70] R. Ostrovsky and B. Patt-Shamir, "Optimal and efficient clock synchronization under drifting clocks," in *Proceedings of the 18th Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '99. New York, NY, USA: ACM, 1999, pp. 3–12. [Online]. Available: http://doi.acm.org/10.1145/301308.301316

[71] A. Pathak, L. Mottola, A. Bakshi, V. K. Prasanna, and G. P. Picco, "Expressing sensor network interaction patterns using data-driven macroprogramming," in *Proceedings of the 5th IEEE International Conference on Pervasive Computing and Communications Workshops*, ser. PERCOMW '07. Washington, DC, USA: IEEE Computer Society, 2007, pp. 255–260. [Online]. Available: http://dx.doi.org/10.1109/PERCOMW.2007.46

[72] C. Perkins and E. Royer, "Ad-hoc on-demand distance vector routing," in *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, February 1999, pp. 90–100. [Online]. Available: http://dx.doi.org/10.1109/MCSA.1999.749281

[73] M. Rahimi, H. Shah, G. Sukhatme, J. Heideman, and D. Estrin, "Studying the feasibility of energy harvesting in a mobile sensor network," in *Proceedings of the IEEE International Conference on Robotics and Automation*, vol. 1, September 2003, pp. 19–24. [Online]. Available: http://dx.doi.org/10.1109/ROBOT.2003.1241567

[74] R. Razavi, J.-F. Perrot, and R. Johnson, "Dart: a meta-level object-oriented framework for task-specific, artifact-driven behavior modeling," in *Proceedings of the 6th OOPSLA Workshop on Domain-Specific Modeling*, 2006, pp. 43–55.

[75] R. Razavi, K. Mechitov, G. Agha, and J.-F. Perrot, "Ambiance: A mobile agent platform for end-user programmable ambient systems," in *Advances in Ambient Intelligence, Frontiers in Artificial Intelligence and Applications*, J. Augusto and D. Shapiro, Eds. Amsterdam, The Netherlands: IOS Press, 2007, vol. 164, pp. 81–106. [Online]. Available: http://portal.acm.org/citation.cfm?id=1563773.1563779

[76] J. Rice, K. Mechitov, S.-H. Sim, T. Nagayama, S. Jang, R. Kim, B. F. Spencer, G. Agha, and Y. Fujino, "Flexible smart sensor framework for autonomous structural health monitoring," *Smart Structures and Systems*, vol. 6, no. 5, pp. 423–438, 2010. [Online]. Available: http://hdl.handle.net/2142/16434

[77] J. Rice, K. Mechitov, S.-H. Sim, B. F. Spencer, and G. Agha, "Enabling framework for structural health monitoring using smart sensors," *Structural Control and Health Monitoring*, 2010. [Online]. Available: http://dx.doi.org/10.1002/stc.386

[78] J. Rice, K. Mechitov, B. F. Spencer, and G. Agha, "A service-oriented architecture for structural health monitoring using smart sensors," in *Proceedings of the 14th World Conference on Earthquake Engineering*, October 2008. [Online]. Available: http://osl.web.cs.illinois.edu/docs/wcee08/wcee08.pdf

[79] O. Salawu, "An integrity index method for structural assessment of engineering structures using modal testing," *Non-Destructive Testing and Condition Monitoring*, vol. 39, no. 1, pp. 33–37, January 1997.

[80] I. Satoh, "Software agents for ambient intelligence," in *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, vol. 2, October 2004, pp. 1147–1152. [Online]. Available: http://dx.doi.org/10.1109/ICSMC.2004.1399778

[81] Seaside Framework. [Online]. Available: http://www.seaside.st/

[82] S.-H. Sim and B. Spencer, "Decentralized strategies for monitoring structures using wireless smart sensor networks," Newmark Structural Engineering Laboratory, University of Illinois at Urbana-Champaign, Tech. Rep. 019, 2010. [Online]. Available: http://hdl.handle.net/2142/14280

[83] M. Singh and M. Huhns, *Service-Oriented Computing: Semantics, Processes, Agents*. John Wiley and Sons, 2005.

[84] A. M.-C. So and Y. Ye, "Theory of semidefinite programming for sensor network localization," in *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, ser. SODA '05. Philadelphia, PA, USA: Society for Industrial and Applied Mathematics, 2005, pp. 405–414. [Online]. Available: http://portal.acm.org/citation.cfm?id=1070432.1070488

[85] H. Sohn, "A review of structural health monitoring literature 1996–2001," in *Proceedings of the 3rd World Conference on Structural Control*, 2003, pp. 9–15.

[86] W. Staszewski, "Structural and mechanical damage detection using wavelets," *Shock and Vibration Digest*, vol. 30, no. 6, pp. 457–472, 1998.

[87] W. Tsai, "Service-oriented system engineering: a new paradigm," in *Proceedings of the IEEE International Workshop on Service-Oriented System Engineering*, October 2005, pp. 3–6. [Online]. Available: http://dx.doi.org/10.1109/SOSE.2005.34

[88] J. Waldo, "Remote procedure calls and Java remote method invocation," *IEEE Concurrency*, vol. 6, no. 3, pp. 5–7, 1998. [Online]. Available: http://dx.doi.org/10.1109/4434.708248

[89] L. Wang and F. Yuan, "Active damage localization technique based on energy propagation of Lamb waves," *Smart Structures and Systems*, vol. 3, no. 2, pp. 201–217, 2007.

[90] K. Whitehouse, F. Zhao, and J. Liu, "Semantic Streams: A framework for composable semantic interpretation of sensor data," in *Wireless Sensor Networks*, ser. Lecture Notes in Computer Science, K. Romer, H. Karl, and F. Mattern, Eds. Springer, 2006, vol. 3868, pp. 5–20. [Online]. Available: http://dx.doi.org/10.1007/11669463_4

[91] J. W. Yoder and R. E. Johnson, "The adaptive object-model architectural style," in *Proceedings of the 3rd IEEE/IFIP Conference on Software Architecture: System Design, Development and Maintenance*, ser. WICSA 3. Deventer, The Netherlands: Kluwer, B.V., 2002, pp. 3–27. [Online]. Available: http://portal.acm.org/citation.cfm?id=646546.693937

[92] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun, "Supporting concurrent applications in wireless sensor networks," in *Proceedings of the 4th International Conference on Embedded Networked Sensor Systems*, ser. SenSys '06. New York, NY, USA: ACM, 2006, pp. 139–152. [Online]. Available: http://doi.acm.org/10.1145/1182807.1182822

[93] C. Zhang, X. Zhou, C. Gao, and C. Wang, "On improving the precision of localization with gross error removal," in *Proceedings of the 28th International Conference on Distributed Computing Systems Workshops*. Washington, DC, USA: IEEE Computer Society, 2008, pp. 144–149. [Online]. Available: http://portal.acm.org/citation.cfm?id=1439272.1439668

[94] F. Zhao, J. Shin, and J. Reich, "Information-driven dynamic sensor collaboration," *IEEE Signal Processing Magazine*, vol. 19, no. 2, pp. 61–72, March 2002. [Online]. Available: http://dx.doi.org/10.1109/79.985685

[95] zlib Compression Library. [Online]. Available: http://zlib.net/