January 2009

Adaptive Middleware for Resource-Constrained Mobile Ad Hoc and Wireless Sensor Networks

Chien-Liang Fok
Washington University in St. Louis

WASHINGTON UNIVERSITY IN ST. LOUIS

School of Engineering and Applied Science

Department of Computer Science and Engineering

Thesis Examination Committee:
Gruia-Catalin Roman, Chair
Chenyang Lu
Tom Bailey
Chris Gill
Christine Julien
Caitlin Kelleher

ADAPTIVE MIDDLEWARE FOR RESOURCE-CONSTRAINED MOBILE AD HOC AND WIRELESS SENSOR NETWORKS

by

Chien-Liang Fok

A dissertation presented to the School of Engineering
of Washington University in partial fulfillment of the
requirements for the degree of

DOCTOR OF PHILOSOPHY

December 2009
Saint Louis, Missouri

ABSTRACT OF THE DISSERTATION

Adaptive Middleware for Resource-Constrained Mobile Ad Hoc and Wireless Sensor Networks

by

Chien-Liang Fok

Doctor of Philosophy in Computer Science

Washington University in St. Louis, 2009

Research Advisors: Professors Gruia-Catalin Roman and Chenyang Lu

Mobile ad hoc networks (MANETs) and wireless sensor networks (WSNs) are two recently-developed technologies that uniquely function without fixed infrastructure support, and sense at scales, resolutions, and durations previously not possible. While both offer great potential in many applications, developing software for these types of networks is extremely difficult, preventing their wide-spread use. Three primary challenges are (1) the high level of dynamics within the network in terms of changing wireless links and node hardware configurations, (2) the wide variety of hardware present in these networks, and (3) the extremely limited computational and energy resources available. Until now, the burden of handling these issues was put on the software application developer. This dissertation presents three novel programming models and middleware systems that address these challenges: Limone, Agilla, and Servilla. Limone reliably handles high levels of dynamics within MANETs. It does this through lightweight coordination primitives that make minimal assumptions about network connectivity. Agilla enables self-adaptive WSN applications via

the integration of mobile agent and tuple space programming models, which is critical given the continuously changing network. It is the first system to successfully demonstrate the feasibility of using mobile agents and tuple spaces within WSNs. Servilla addresses the challenges that arise from WSN hardware heterogeneity using principles of Service-Oriented Computing (SOC). It is the first system to successfully implement the entire SOC model within WSNs and uniquely tailors it to the WSN domain by making it energy-aware and adaptive. The efficacies of the above three systems are demonstrated through implementation, micro-benchmarks, and the evaluation of several real-world applications including Universal Remote, Fire Detection and Tracking, Structural Health Monitoring, and Medical Patient Monitoring.

Acknowledgments

I would like to thank my advisors Gruia-Catalin Roman and Chenyang Lu for their invaluable advice and support.

Chien-Liang Fok

Washington University in Saint Louis
December 2009

Contents

Abstract . . . . . **ii**

Acknowledgments . . . . . **iv**

List of Tables . . . . . **viii**

List of Figures . . . . . **ix**

1 Introduction **1**
1.1 Challenges Addressed 7
1.2 Dissertation Overview 7

2 Background **9**
2.1 Targeted Network Platforms 9
2.1.1 Mobile Ad Hoc Networks 9
2.1.2 Wireless Sensor Networks 11
2.2 Coordination Techniques 12
2.2.1 Tuple Spaces 13
2.2.2 Mobile Agents 14
2.2.3 Service-Oriented Computing 15

3 Limone: A Lightweight Coordination Model for Mobile Ad Hoc Networks **16**
3.1 Motivation 17
3.1.1 Application Example: Universal Remote 19
3.2 Programming Model 22
3.2.1 Reactive Programming: Tuple Space 25
3.3 Middleware Implementation 26
3.4 Microbenchmarks 33
3.5 Application Case Study: Universal Remote 34
3.6 Chapter Summary 37

4 Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks **38**
4.1 Motivation 39
4.1.1 Adaptation through In-network Reprogramming 42
4.2 Programming Model 43

4.2.1 Mobile Agents . . . 45
4.2.2 Tuple Space . . . 45
4.2.3 Location-Based Addressing . . . 48
4.2.4 Example . . . 49
4.2.5 Scalability . . . 50
4.2.6 Adaptation to Node Failures . . . 50
4.2.7 Security . . . 51
4.3 Implementation . . . 51
4.3.1 Target Platform . . . 51
4.3.2 Middleware Architecture . . . 52
4.3.3 Agent . . . 54
4.3.4 Instruction Set Architecture (ISA) . . . 55
4.4 Micro-benchmarks . . . 57
4.4.1 Micro-Benchmarks . . . 57
4.5 Application Case Study: Fire Detection . . . 63
4.6 Application case study: Robot Navigation . . . 68
4.7 Agimone: Integrating Wireless Sensor Networks with IP Networks . . . 70
4.7.1 System Architecture . . . 72
4.7.2 Migration Across WSNs . . . 74
4.7.3 Evaluation . . . 74
4.7.4 Application Case Study: Cargo Container Monitoring . . . 80
4.8 Chapter Summary . . . 81

5 Servilla: Service Provisioning for Wireless Sensor Networks . . . 83
5.1 Motivation . . . 84
5.2 Related Work . . . 86
5.3 Programming Model . . . 88
5.3.1 Service Binding . . . 91
5.3.2 Novel Binding Semantics . . . 93
5.3.3 Service Invocation . . . 94
5.4 Programming Languages . . . 95
5.4.1 ServillaSpec . . . 96
5.4.2 ServillaScript . . . 99
5.5 Middleware Architecture and Implementation . . . 104
5.5.1 SPF-Consumer . . . 104
5.5.2 SPF-Provider . . . 106
5.5.3 Middleware Modularity . . . 107
5.6 Micro-benchmarks . . . 111
5.6.1 Memory Footprint . . . 111
5.6.2 Efficiency of Service Binding . . . 112
5.6.3 Efficiency of Service Invocation . . . 115
5.7 Application Case Study: Structural Health Monitoring . . . 117
5.8 Chapter Summary . . . 125

6 Servilla Extension: Adaptive Service Provisioning . . . 130
6.1 Motivation . . . 131
6.2 Related Work . . . 132
6.3 Problem Definition . . . 135
6.3.1 System Model . . . 135
6.3.2 Design Goals . . . 136
6.4 Adaptation Mechanisms . . . 138
6.4.1 Energy-Aware Provider Selection . . . 138
6.4.2 Efficiency Through Invocation Sharing . . . 143
6.4.3 Adapting to Network Topology Changes . . . 146
6.5 Evaluation . . . 149
6.5.1 Energy Efficiency when Idling . . . 150
6.5.2 Energy Efficiency of Wireless Transmission . . . 154
6.5.3 Energy Efficiency of Wireless Reception . . . 158
6.5.4 Energy Efficiency of Sensing . . . 160
6.6 Applications . . . 161
6.6.1 Medical Patient Monitoring . . . 162
6.6.2 Structural Health Monitoring . . . 167
6.7 Chapter Summary . . . 169

7 Future Work . . . 170

8 Conclusions . . . 173

Appendix A Measuring the Energy Consumption of WSN Devices 175

Appendix B Derivation of the Energy Utilization Equations for the Structural Health Monitoring AccelTrigger Service . . . 177
B.1 Local Invocation . . . 178
B.2 Remote Invocation . . . 179
B.3 Local vs. Remote Invocation . . . 180
B.4 Example Scenario 1 . . . 181
B.4.1 Energy-Aware Calculations . . . 181
B.4.2 Validation of Equations . . . 184

References . . . 185

Vita . . . 202

List of Tables

4.1 Memory Availability and Size of Agilla 51

5.1 Various service binding semantics and when they should be used. . . 93
5.2 WSN devices vary widely in computational resources. 108
5.3 The size of the properties within service specification `FFT` 113
5.4 Service matching latency when comparing two FFT-real service specifications 113
5.5 The sizes of the specifications used to evaluate service invocation . . . 115
5.6 The latency of obtaining the a service's binding state 116
5.7 Power and latency attributes of TelosB and Imote2 platforms when radio is operating at 1% duty cycle. 121

6.1 Variables for deriving the energy cost of service invocation, and who must supply them. 140
6.2 The timing and power attributes of sending one acknowledged packet. The numbers are obtained using an oscilloscope and averaged over ten packet transmissions. The average and 95% confidence intervals are shown. 157
6.3 The latency and power attributes of receiving a packet. 159
6.4 The timing and power attributes of sensing. 161
6.5 The success rate of service invocation of the medical patient monitoring application. 164
6.6 The average number of beacons transmitted per invocation over all experimental rounds. 165

List of Figures

3.1 The Limone model. Software agents are represented by ovals. Each agent owns a local tuple space (LTS) and an acquaintance list (AQL). Agent C is shown migrating to host Y. The dotted rectangle surrounding the tuples spaces of agents B, C, and D highlight those that are accessible by agent C. 23
3.2 The overall structure of Limone. 26
3.3 Acquaintance list. 27
3.4 Local tuple space operations. 29
3.5 Operations on a remote tuple space. 30
3.6 Reaction Registry. 31
3.7 Reaction List. 33
3.8 Application code size and round-trip message passing time using reactions as a trigger, averaged over 100 rounds. 34
3.9 The Universal Remote Application's User Interface 35

4.1 The Agilla model. Each node in the network maintains a node neighbor list, a discrete local tuple space, and multiple mobile agents. The mobile agents are able to migrate between nodes, and access the tuple spaces belonging to remote nodes. 44
4.2 A portion of the **FireTracker** agent 49
4.3 Agilla's middleware architecture 52
4.4 Messages used during migration 53
4.5 The mobile agent architecture 55
4.6 Noteworthy Agilla instructions 56
4.7 The agents that test `smove` (top) and `rout` (bottom) 58
4.8 `smove` vs. `rout` reliability 58
4.9 `smove` vs. `rout` latency 58
4.10 Remote operation lantency 60
4.11 Local operation latency 60
4.12 Test program pseudocode 61
4.13 The layout of the WSN testbed consisting of 31 TelosB nodes spread across 1000 square meters. Red 'x' indicate node placement, the green 'o' marks the gateway. 62
4.14 Maté and Agilla reprogramming rates 62
4.15 Deluge reprogramming rate 62

4.16 An overview of the fire detection and tracking application. When a fire breaks out, detection agents sense the fire (1) and send a message to a base station (2), which injects a tracker agent into the network (3). This agent migrates to the fire and clones itself to form a perimeter. The perimeter is continuously adjusted based on the fire's behavior. . 63
4.17 The static `Fire` agent 64
4.18 A `FireDetector` agent 64
4.19 The reaction registered by the `FireTracker` agent 65
4.20 The life cycle of a `FireTracker` agent 66
4.21 The static fire test scenario, the star is the initial position of the `FireTracker` agent 66
4.22 The rate of forming a perimeter around static fires 67
4.23 Dynamic Fire Test Settings 68
4.24 Dynamic Fire Perimeter Formation 68
4.25 The robot navigation problem. A roadmap graph is overlaid on the WSN and mobile agents are used to query the temperature along the edges, which helps the robot navigate around the fires. 69
4.26 The Agimone system architecture. Each WSN has a gateway running Limone. The gateways communicate using Limone, forming a bridge for Agilla agents to migrate between WSNs. A Limone registry is used to discover available WSNs 71
4.27 Agilla Agent Migration Across Different WSNs 73
4.28 The Latency of Remote Tuple Space Operations 75
4.29 The Five Stages of an Inter-WSN Agent Migration Operation. 76
4.30 The Latency of Each Agent Migration Stage (Average of 1000 Runs) 76
4.31 The In-and-Out Agent Migration Latency. 78
4.32 The End-to-End Migration Latency 78
4.33 End-to-End Latency vs. Size of Agent 79

5.1 Servilla targets heterogeneous WSNs in which different classes of devices provide services that are used by application tasks either locally or remotely. Services are platform-specific while tasks are platform-independent. 89
5.2 A specification describing a FFT service 96
5.3 Possible ambiguity: Does attribute Error modify the Latency attribute or the readx output? 98
5.4 ServiceSpec specifications are extensible using the import keyword . . 99
5.5 A task that invokes an accelerometer-sensing service 10 times 100
5.6 The **uses** keyword allows a script to use multiple services that have the same specification. 100
5.7 The **error** keyword specifies an error callback function that is executed when the invocation fails. 102

5.8 The **error** keyword specifies an error callback function that is executed when the invocation fails. 103
5.9 Servilla's middleware consists of a virtual machine and a service provisioning framework (SPF). The SPF consists of a consumer and provider. 104
5.10 The detailed architecture of the Service Provisioning Framework. . . . 105
5.11 Servilla's middleware components. 109
5.12 All services must provide this interface. 110
5.13 The code memory footprint of different Servilla configurations on the TelosB platform. 111
5.14 The latency of comparing a specification. 115
5.15 The latency of obtaining a service's binding state. 115
5.16 The services used by the damage localization application 118
5.17 The damage localization application task using on-demand invocations 126
5.18 The damage localization application task using event-based invocations 127
5.19 Percent power savings of heterogeneous vs. homogeneous WSNs. . . . 128
5.20 Relative power savings of different invocation types on heterogeneous WSNs. 129

6.1 The actions performed during periodic invocations. 139
6.2 The actions performed during event-based invocations. 141
6.3 A visualization of how service utilization is calculated. 143
6.4 A naïve brute-force method for calculating utilization. 145
6.5 A algorithm for calculating service utilization when service sharing is possible. 146
6.6 A finite state machine capturing the behavior of the adaptation mechanism used to adapt to network topology changes. 147
6.7 Measured and theoretical P_{idle} of Imote2 and Telosb devices 151
6.8 The power draw of the TelosB and Imote2 when idling with the radio on and off. 152
6.9 The power draw of an Imote2 when it transmits 5 packets. 155
6.10 T_{search} versus the duty cycle, both actual and theoretical. The results indicate that, on average, T_{search} is half of the duty cycle period. . . . 156
6.11 The power draw of a TelosB receiving 5 packets. 158
6.12 The power draw of taking an accelerometer reading. 160
6.13 A map of the WSN testbed used in the medical patient monitoring application. The testbed nodes provide relay services for delivering medical patient data to the base station, which is represented as a red triangle. The dotted lines marks the $358.71m$ route the patient traveled during each experimental round. 162
6.14 The average number of messages transmitted per invocation. 165
6.15 The latency of invoking the relay service. 166

6.16 The predicted and actual energy footprints of the structural health monitoring application scenario when `DutyCycle` = 10 and `InvokePeriod` = 1000 in. 168

A.1 The circuit used to measure the power draw of a WSN device. Two probes from the same oscilloscope simultaneously measure voltages $V1$ and $V2$ at junctions J2 and J3, respectively. Both are grounded at junction J1. $V1$ measures the voltage across resistor R1 and is used to calculate the instantaneous current, $i = \frac{V1}{R1}$. $V2$ measures the voltage across the WSN device. The power, P, of the WSN device is thus $P = i \cdot V2$ 176

Chapter 1

Introduction

Wireless sensor networks (WSNs) and Mobile Ad hoc Networks (MANETs) are two unique forms of networking that have the potential to make a significant impact in our daily lives. WSNs consist of a multitude of tiny devices embedded in the environment that are capable of sensing, computation, and communication. They revolutionize the capabilities of certain critical applications like tracking and monitoring by enabling higher density sensing at significantly lower cost. MANETs consist of mobile devices like laptops, netbooks, cellphones, and PDAs that are capable of wireless communication. MANETs differ from traditional networks in that there is no hierarchy between devices. Instead, devices opportunistically form peer-to-peer wireless links whenever they come within range and break the links whenever they move out of range. They enable networks to form in situations that would otherwise not be possible and have many important applications like coordinating first responders in a disaster scenario where the networking infrastructure is destroyed. As relatively new and unique technologies, existing software engineering techniques, programming models, and middleware do not adequately address the many novel and fundamental challenges presented by these networks. Chief among these are (1) the need for a lightweight minimalist framework for facilitating the development of reliable applications in highly dynamic and mobile environments, (2) the need for applications to be self-adaptive in a changing environment, (3) the need to carefully manage the minimal resources available in WSNs, (3) the need to integrate WSNs with the existing computer network infrastructure, (4) the need to support device heterogeneity and network dynamics, and (5) the need for adaptive service provisioning in such resource-constrained and dynamic environments. Throughout this dissertation, new software

engineering techniques and programming models were developed that specifically address each of the above challenges. In addition, middleware frameworks, services, and application prototypes were implemented to demonstrate the efficacy of our solutions. Each of these is now presented.

(1) Developing a lightweight coordination middleware for MANETs. The highly transient and unpredictable nature of wireless links within MANETs make reliable application development extremely difficult. To address this difficulty, numerous powerful middleware systems were created that deal with the underlying dynamics within the network. These systems introduced many interesting and novel constructs that make network disconnections more predictable, like instituting a "safe zone" smaller than the actual wireless range, and only communicating with devices within the safe zone [152]. While these systems had strong theoretical underpinnings and provide many valuable guarantees, the level of services and guarantees provided were more than what most MANET applications required. For example, the main motivation behind instituting the aforementioned safe zone is to enable distributed transactions among groups of devices. This is not necessary in applications that only involve communication between at most two users since the wireless link connecting the two devices form and break atomically. To address this, Limone took a different approach towards addressing the challenges imposed by MANETs. Instead of trying to provide powerful consistency and atomicity guarantees in a dynamic environment, Limone sought to find the minimalist set of constructs that are useful to an application developer, while making *no* assumptions about wireless link behavior. Starting with the most basic assumption that a single round-trip communication with a remote device is eventually possible, Limone sought to build upon this a middleware that is adaptive and resilient to unpredictable changes in the underlying network. The investigation resulted in a new and unique coordination model that was unlike any of the previously existing models, and was better in the sense of being able to operate in a wider range of environments.

(2) Supporting Self-Adaptive Applications in WSNs. WSN applications must be self-adaptive due to the continuously changing environment in which the network is deployed. Since WSN nodes are embedded and can sense the environment, changes within the environment impact the network, both in terms of the set of applications that should execute, and the way the applications should behave. For example, a

WSN deployed in a forest may initially be used for habitat monitoring. Later, when the probability of fire is high, the network may be used for fire detection. If a fire breaks out, the network would best be used to track the fire. Creating software that is flexible enough to satisfy the diverse requirements of a WSN is challenging, especially given the limited amount of resources available within a WSN. To address this, Agilla, a middleware that facilitates self-adaptive WSN applications, was developed. Agilla is among the first WSN middleware platforms to offer both mobile agent and tuple space programming models for developing applications. Mobile agents are special software processes that can explicitly clone or migrate across WSN nodes. They can do this while maintaining their state, thus elegantly capturing computations that execute across multiple WSN nodes. Tuple spaces offer a shared memory space in which data elements are accessed via pattern matching, allowing independently-developed mobile agents to freely migrate while still being able to communicate. By merging these two programming models into a WSN middleware platform, Agilla enabled WSN applications to restructure themselves in response to a changing environment. To demonstrate this, we used Agilla to implement a fire detection and tracking application that dynamically adjusts itself to maintain a perimeter around a spreading wildfire. As the wildfire spreads, it disables WSN nodes and eventually breaks the perimeter. Once broken, the mobile agents adjacent to the breakage clone themselves around the hole, thus maintaining the integrity of the perimeter.

Tracking is another critical application of WSNs. Tracking is challenging due to the dynamic nature of the phenomenon being tracked. The application must adapt whenever the phenomenon moves or changes. Agillas programming model is useful in developing tracking applications that are able to adapt to changes in the phenomenon being tracked. To demonstrate this, we used Agilla to implement a cargo container tracking application. Tracking cargo containers is important for national security and logistical reasons. It is complex due to the continuous movement, rearrangement, and exchanging of cargo containers between different administrative domains as they travel around the world. To secure and track cargo containers, a wireless sensor network can be deployed such that each shipping container contains a WSN node. This is demonstrated by an application called AgiTrack, which was implemented on top of Agilla. Using Agillas flexible programming model, several diverse tasks are demonstrated including counting the number of containers, searching for items within the containers, and securing containers. By implementing AgiTrack using

mobile agents and tuple spaces, it is able to continue to execute seamlessly despite reconfigurations in the container orientations.

(3) Integration of WSNs and Internet Protocol (IP) Networks. The aforementioned middleware and applications run within the confines of a single WSN. This typified many early WSN applications as the new and unique characteristics of WSNs like their emphasis on energy-efficiency prevented them from being integrated with the existing network infrastructure like the Internet, which is based on the Internet Protocol (IP). This is unfortunate since networks like the Internet offer unmatched connectivity, enabling near-universal access, and tremendous resources in terms of computing and data storage. Integrating sensor and IP networks into a uniform platform enables applications to take advantage of the resources available on traditional IP networks while still receiving sensor data obtained from within a sensor network. Providing a unified platform that spans both types of networks would also facilitate flexible application deployment. To this end, we developed a software middleware framework called Agimone that allows applications to be deployed on a WSN in the form of mobile agents, which can autonomously discover and migrate to other WSNs using a common IP backbone as a bridge. Agimone was the first system to allow mobile agents to migrate between sensor and IP networks. It facilitated data sharing between WSNs and the IP network through remote tuple space operations. Using this framework, computationally weak sensing nodes could defer expensive computations to more-powerful devices. To evaluate Agimone, it was used to re-implement AgiTrack, the cargo-tracking application described previously. Micro-benchmarks on the latency of Agimone operations demonstrated feasibility and applicability to many applications.

(4) Supporting Platform Heterogeneity in Wireless Sensor Networks. Another important consequence of integrating WSNs and the Internet is that it is only a matter of time before the diversity of devices that constitute the network grows to enormous proportions. WSN heterogeneity is a major hurdle in the development and deployment of WSN applications. This is primarily due to the limited resources available, which require that applications be carefully engineered in platform-specific ways for maximum efficiency. Explicitly engineering an application in a platform-specific manner is labor-intensive, error-prone, and unlikely to result in software that can run on other platforms or optimization techniques that can be applied to other

applications. To solve this problem, a programming model that had yet to be entirely used in WSNs — Service Oriented Computing (SOC) — is used. SOC is a powerful programming model in that it provides a loose coupling between software components. This enables software components written by different organizations to function together seamlessly. SOC principles are applied to WSNs by using them as a separation between platform-independent application scripts and platform-specific services. Using SOC, the scripts would be automatically and dynamically bound to services, based on the specific characteristics of the hardware that is available. Applications, being platform-independent, could execute anywhere regardless of the type of hardware available, thus simplifying application development. Services, being platform-specific, provide access to specialized capabilities of the underlying hardware and are tailored to maximize energy efficiency. The mechanisms for achieving SOC in WSN were integrated into a new middleware system called Servilla, the first service-oriented programming framework to function entirely within a WSN. Servilla integrates aspects of service provisioning and scripting and tailors them to the unique properties of WSNs to enable applications that are platform-independent and yet able to access platform-specific functionality. Specifically, scripting enables applications to be platform-independent by executing within a virtual machine, while service provisioning enables applications to efficiently access platform-specific resources. Through an evaluation on a heterogeneous WSN consisting of TelosB and Imote2 nodes involving a structural health monitoring application, Servilla demonstrated the feasibility of using these programming models within WSNs and the efficacy of using them to develop platform-independent applications that can still efficiently access platform-specific resources.

(5) Adaptive Service Provisioning in Wireless Sensor Networks. WSNs are extremely dynamic systems requiring that the application continuously adapt to a changing network topology and resource levels. SOC provides a natural decoupling between applications and resources within the network. Since SOC was already being used in WSNs for the purpose of handling network heterogeneity, it could be easily modified to become adaptive to changes in the network. The two key ways in which SOC is made adaptive within WSNs is in the ability to automatically switch providers if the current provider fails, and the consideration of energy constraints in the selection of a new provider. Automatically switching providers is important because it enables the middleware to address challenges that arise from a changing network topology.

SOC turns out to be a perfect mechanism for achieving this form of adaptation since it already decouples service consumers from providers, thus enabling the seamless switching of providers from the consumer's perspective. Energy is a fundamentally scarce resource in most WSNs since nodes are typically powered by battery. Making service provisioning energy-aware helps conserve energy resources, enabling prolonged network lifetime.

The above discussion highlights the various forms of software engineering techniques, programming models, middleware, services, and applications for WSNs and MANETs that are discussed in this dissertation. The research initially focused on ways to lightly coordinate MANET applications, but soon focused on how to simplify application development, while enabling applications to be more flexible and adaptive. The work described resulted in the integration of WSNs with each other and the Internet, forming larger and more complex systems consisting of many types of devices and which multiple applications must share. To this end, new programming models, middleware, and services were developed that assist developers in creating applications that can handle device heterogeneity, efficiently allocate resources, and manage the network. Finally, the most recent activities include investigations into how to address network dynamics. WSNs are relatively dynamic given their limited resources and exposure to potentially harsh and continuously changing environments. This dynamic nature should be reflected in the bindings between applications and services since the set of services that are available and the wireless link quality between the application and its services are continuously changing. Adaptive service provisioning is critical since selecting the correct service provider may result in significant energy savings and improvement in quality of service. The key mechanisms for enabling adaptation and the additional parameters and equations necessary to perform energy calculations are identified and presented. Furthermore, since service discovery and binding are done by the middleware, the adaptation mechanisms are hidden from the application, resulting in little to no increase in application complexity.

1.1 Challenges Addressed

The challenges addressed in this dissertation are five-fold. First, the issue of creating the minimalist useful coordination model for MANETs is addressed. This required carefully selecting the appropriate set of coordination primitives and operational semantics so as to ensure minimal assumptions about the underlying network.

Second, the issue of facilitating adaptive applications in WSNs is addressed. This involved implementing, for the first time, support for mobile agents and tuple spaces within WSNs.

Third, the issue of integrating WSNs with traditional networks is addressed. This was done by integrating the two aforementioned middleware systems for MANETs and WSNs, respectively.

Fourth, the issue of how to adapt to network heterogeneity is addressed. This was done by bring in to the WSN domain, for the first time, the entire SOC programming model. Using SOC, applications could be written in a platform-independent manner while still being efficient.

Fifth, the issue of enabling adaptive energy-aware SOC within WSNs is addressed. This is important because selecting the "right" set of services within the WSN can make a big difference in terms of energy consumption.

1.2 Dissertation Overview

The dissertation is organized as follows. Chapter 2 provides background information. Chapter 3 presents Limone, the lightweight coordination model for facilitating adaptive applications in MANETs. Chapter 4 presents Agilla, the first mobile agent middleware for WSNs. It demonstrates how WSN applications can be made adaptive and WSNs can be seamlessly integrated with IP networks using mobile agents and tuple spaces. Chapter 5 presents Servilla, the first middleware to fully utilize the SOC coordination model within WSNs enabling in-network collaboration between WSN devices. It describes how SOC is used to handle network heterogeneity by enabling

applications to be platform-independent while still able to access platform-specific functions. Chapter 6 presents how Servilla is extended to increase energy efficiency by judiciously adjusting the bindings between services and applications, and service availability by automatically switching providers when connectivity to the current provider breaks. Future work is presented in Chapter 7. The dissertation ends with conclusions in Chapter 8.

Chapter 2

Background

2.1 Targeted Network Platforms

The targeted network platforms of the middleware systems described in this dissertation include mobile ad hoc networks (MANETs) and wireless sensor networks (WSNs). Both these types of networks are made possible by recent advances in technology, most notably in the areas of device miniaturization, batteries, and wireless communication. Unlike traditional networks, MANETs and WSNs are *ad hoc*, meaning they consist of devices that form peer-to-peer wireless links directly between each other, rather than through a wireless base station that is part of and connected to the wired network infrastructure. The advantage of wireless ad hoc networks is the ability to form without fixed infrastructure support, enabling deployment in situations previously not possible. WSNs are a special type of MANET in which the network devices are embedded in and can sense the environment. Each of these types of networks are now discussed.

2.1.1 Mobile Ad Hoc Networks

Mobile devices with wireless capabilities have experienced rapid growth in recent years due to advances in technology and social pressures from a highly dynamic society. These devices include laptops, netbooks, cell phones, PDAs, and even some watches. In addition to communicating with infrastructure-based networks like cellular and WiFi hotspots, many of these devices are capable of forming ad hoc networks, which

are networks that form directly between devices with no central coordinator or fixed infrastructure support. Ad hoc networks are formed opportunistically by the chance encounter of two devices supporting the same wireless interface. The simple act of moving within communication range results in a wireless link through which the two devices may communicate. By eliminating the reliance on the wired infrastructure, ad hoc networks can be rapidly deployed in disaster situations where the infrastructure has been destroyed, or in military applications where the infrastructure belongs to the enemy. Ad hoc networks are also convenient in day-to-day scenarios where the duration of the activity is too brisk and localized to warrant the establishment of a permanent network infrastructure.

Applications for ad hoc networks are many. As previously mentioned, a primary benefit of MANETs is their ability to function without fixed infrastructure support. Thus, any application in which the fixed infrastructure is damaged or non-existent is a potential candidate for MANETs. Typical examples include coordinating first responders that arrive at a disaster location where the fixed networking infrastructure is destroyed, enabling peer-to-peer communication among military units deployed in a hostile region where the infrastructure does not exist, facilitating quick exchange of data like business contact information among people who are in close physical proximity for short periods of time, and playing multi-player games in which each player holds a device that communicates with every other device via the ad hoc network, enabling the players to move about freely while still coordinating their actions. The number of applications for MANETs is expected to grow as more powerful middleware systems are developed that simplify the creation of applications for mobile ad hoc networks.

The salient properties of MANETs create many challenges for the application developer. The inherent unreliability of wireless signals and the mobility of nodes result in frequent unannounced disconnections and message loss. In addition, mobile devices have limited battery and computing power. The limited functionality of mobile devices and the peer-to-peer nature of the network lead to strong mutual dependencies among devices, which have to cooperate to achieve a variety of common goals. This results in an increased need for coordination support. For example, in a planetary exploration setting, miniature rovers, each equipped with a single sensor, may need to perform experiments that demand data from any arbitrary combination of sensors.

Middleware systems are often used as a mechanism for addressing the challenges of programming software for environments that would otherwise be difficult to program in. The three systems presented in this dissertation focus on one aspect of why developing applications for ad hoc networks is difficult — the level of dynamics present in such networks.

2.1.2 Wireless Sensor Networks

Wireless Sensor Networks (WSNs) [43] are a special class of MANETs in which the devices contain sensors that can gather data about the environment and are typically embedded in the environment for long periods of time (months to years). A WSN node is remarkably small. Many of them are approximately the size of a matchbox, through some are significantly smaller [174]. The primary goal of WSNs is to sense the environment in which they are embedded.

There are many applications for WSNs [113, 30, 17, 108, 183, 75, 104, 6]. They include habitat monitoring on the Great Duck Island [113] and in the James Reserve [30], and microclimate research around redwood trees [17], surveillance, medical care [108], structural integrity monitoring [183], highway automation [75] and military operations [104].

Since WSN nodes are embedded, each device is typically small, battery powered, and communicates over low-power unreliable wireless radios. Most current WSN devices differ from MANET devices in terms of amount of computation, energy, and network bandwidth resources available, often by several orders of magnitude. WSNs may be ad hoc where they autonomously form a network without infrastructure support. Depending on the application, the network may form routing trees for delivering data to base stations, or a multi-hop mesh for delivering data amongst themselves.

WSNs also have the same challenges that face MANETs. Since the devices communicate over low power wireless radios, the wireless links with WSNs are also very dynamic, requiring that nodes adapt to the set of nodes that are within communication range. In addition, since WSNs are embedded in the environment, a node

can be easily damaged, stolen, or disabled, further contributing to the dynamic network topology. Another aspect that is particularly acute for WSNs is the amount of resources available, in terms of computational ability, energy, and wireless bandwidth. WSN nodes are physically smaller than typical MANET devices, resulting in nodes that have very limited processing power, energy, and wireless communication capabilities. This lack of resources makes developing applications significantly more difficult, motivating the use of middleware and coordination techniques for simplifying application development.

2.2 Coordination Techniques

Mechanisms that address the complexities of ad hoc networks include enhancements to the operating system, specialized languages, and middleware. Among these, middleware is the most popular. Operating systems are tightly integrated with low-level communication services and expose too many details that complicate the programming tasks. The development and use of new programming languages is costly and entails great risks. Middleware, however, provides high-level abstractions while minimizing risk by leveraging off the existing software infrastructure. When designed properly, middleware can divert attention from mundane concerns like low-level protocol development, to more fruitful areas involving application-specific goals.

Designing a coordination middleware for ad hoc networks is difficult. It must be lightweight in terms of the amount of power, memory, and bandwidth consumed. Depending on the application, it may have to operate over a wide range of devices with different capabilities: some devices, such as a laptop, may have plenty of memory and processing ability, while others, such as a node in a sensor network, may have extremely limited resources. A coordination middleware for ad hoc networks must be flexible in order to adapt to a dynamic environment; for example, in a universal remote control application, a remote held by the user must interact with a set of devices within its vicinity, a set that changes as the user moves. Furthermore, wireless signals are prone to interference from the environment. Thus, the middleware must be designed to handle unpredictable message loss.

Coordination middleware facilitates application development by providing high-level constructs such as tuple spaces [57], blackboards [50], and channels [122, 123], in place of lower-level constructs such as sockets. Tuple spaces and blackboards are both shared-memory architectures. Tuple spaces differ from blackboards in that they use pattern-matching for retrieving data; in a blackboard, the data is generally accessed by type alone. Channels are similar to sockets in that data is inserted at one end and is retrieved from the other. They differ in that the end points of a channel may be dynamically rebound.

These high-level constructs facilitate coordination by providing a layer of decoupling between nodes. In order to create a socket, the identity of the destination must be known and remains fixed. This is rather inflexible and complicates application development, particularly in ad hoc networks where connectivity is dynamic. High-level constructs, however, do not require the sender and receiver to be aware of each other. When using a tuple space or blackboard, the node that inserts data need not know the node that later extracts it. Also, since the shared space is public, multiple nodes may retrieve the same data. When using a channel, the sender need not know which node is bound to the receiving end of the channel. This level of decoupling simplifies application development because changes in connectivity no longer need to be dealt with explicitly.

2.2.1 Tuple Spaces

Tuple spaces [57] are a form of shared memory in which data elements, called *tuples*, are accessed using pattern matching instead of direct memory address. They provide standard operations like `out` (insert a tuple), `in` (remove a tuple), and `read` (read a tuple). The main advantage of using tuple spaces is the *decoupling* it provides among different communicating software components. For example, one component may insert a tuple and leave. Later, another component unknown to the first may arrive and receive the tuple. Thus two components are communicating without actually being aware of each other's presence. Using a tuple space, the communication is able to occur regardless of time and space, a phenomena called *spatio-temporal decoupling.* This decoupling is important in networks that are highly dynamic since they enable communication despite changes in the underlying network topology.

While a tuple space is logically perceived to be a single shared memory space, it may in fact be physically distributed among multiple nodes in the network. When a tuple space is distributed among multiple nodes, it takes the form of a single logical "federated tuple space" in which the contents appear local but are actually located on different devices. A key benefit of using tuple spaces is the fact that whether the tuple space is local or federated is hidden from the applications. This is important because changes in the underlying network topology can be hidden from the application developer, thus simplifying applications.

2.2.2 Mobile Agents

Mobile agents are special software processes that have the capability of migrating from one node to another while maintaining their execution state. This results in the ability to carry out a sequence of computations that span multiple nodes in the network. Mobile agents have been used in the Internet [2, 18, 15, 29, 42, 61, 83, 139, 140] and their potential benefits are well established [94, 112, 182, 148, 168, 169, 181]. Some systems for the Internet include Agent Tcl [61], Ara [140], Java Aglets [139], Mole [18], Sumatra [2], TACOMA [83] PeerWare [42], and MARS [29]. They have been used in data mining [94], e-commerce [112], and network management applications [15]. Since these systems are designed to run on Internet servers, where computational resources are relatively plentiful and the links relatively static, efficient resource utilization is not their main focus.

A few important aspects of mobile agents are worth mentioning. First, mobile agents are just like any other software process except for their ability to migrate to a different host. Thus, they are a unit of execution analogous to a thread or process within an application. When they migrate, they usually carry with them both their code and state, enabling them to continue executing where they left off. Mobile agents can optionally not carry their state, a process called *weak migration*, which requires that the agent restart from the initial state upon arriving at the destination.

The ability for mobile agents to migrate across nodes is powerful and enables greater degrees of flexibility relative to traditional statically-installed code. This flexibility is

exploited in the design of Limone and Servilla for the purpose of increasing application adaptability to a changing and highly dynamic environment.

2.2.3 Service-Oriented Computing

Service-Oriented Computing (SOC) [137] is a programming model that consists of service consumers, providers and a service registry. Its primary advantage stems from the *decoupling* of the consumers and providers, enabling them to be developed by different organizations. Specifically, consumers and providers each submit *service specifications* that describe the service required or offered, and are used by the service-oriented architecture (SOA) to automatically *match* and *bind* consumers to providers.

SOC enables loose coupling between service consumers and providers through service descriptions that can be automatically compared and matched. This decoupling enables independently-developed applications to work together. For example, it enables an Internet-based application running on a webserver to access data produced by another application executing within a WSN.

This dissertation uses SOC for a slightly different purpose – to simplify WSN applications by enabling them to be platform-independent, adaptive, and energy efficient. Using SOC, WSN applications can be service consumers that are dynamically bound to services provided by the underlying hardware. This enables platform-independent applications since platform-specific functionalities can be accessed through services, simplifying programing. Since services are provided by the hardware, they can be optimized enabling higher degrees of efficiency. The dynamic binding between service consumers (the applications) and providers (the hardware) is the key enabler that one of the middleware platforms presented in this dissertation exploits (Servilla), to allow WSN applications to handle network heterogeneity.

Chapter 3

Limone: A Lightweight Coordination Model for Mobile Ad Hoc Networks

Limone (Lightly-Coordinated Mobile Networks) is a novel coordination model and middleware that facilitates application development in MANETs. It targets dynamic ad hoc networks in which communication links are transient and unpredictable by using lightweight coordination primitives that make minimal assumptions about the execution environment. Specifically, no knowledge about when wireless links form or break is assumed. Instead, the model starts with the premise that a single round-trip message exchange is possible and, under this minimalist assumption, offers a reasonable set of lightweight primitives with precise functional guarantees. Using this set of lightweight primitives, Limone enables MANET applications to be developed quickly and reliably. The willingness to accommodate a high degree of uncertainty about the physical state of the system raises important research questions regarding the choice of coordination style and associated constructs. A minimalist philosophy, combined with the goal of achieving high levels of performance, led to the emergence of a novel model whose elements appear to support fundamental coordination concerns. Central to the model is the organization of all coordination activities around an *acquaintance list* that reflects the current local view of the global operating context, and whose composition is subject to customizable admission policies. From the application's perspective, all interactions with other components take place by referring to individual members of the acquaintance list. All operations are content-based, but can be active or reactive. This perspective on coordination, unique to Limone, offers an

expressive model that enjoys an effective implementation likely to transfer to many MANET environments. This chapter introduces Limone, explains its key features, and explores its capabilities as a coordination model. To provide a concrete illustration of the model and its applications, a universal remote application is used as a running example.

3.1 Motivation

Mobile devices like cellphones and laptops with wireless capabilities have experienced rapid growth in recent years due to advances in technology and demands from a highly mobile society. Many of these devices are capable of forming MANETs, in which they communicate directly with neighboring devices via peer-to-peer wireless links. By not relying on a fixed infrastructure like physical wires or wireless base stations, MANETs can be rapidly deployed in disaster situations where the infrastructure has been destroyed or in search-and-rescue scenarios in remote locations where infrastructure does not exist. MANETs are also convenient in day-to-day scenarios where the duration of the activity is too brisk, localized, and transient to warrant the establishment of a network infrastructure. Applications for MANETs are important because they are able to execute in environments in which traditional infrastructure-based applications cannot.

The salient properties of MANETs create many challenges for application developers. The inherent unreliability of wireless signals and the mobility of nodes result in frequent and unannounced disconnections, which can lead to numerous problems including message loss, data loss, and application deadlock. In addition, mobile devices typically have limited resources in terms of energy, computing capability, and communication bandwidth. The limited functionality of mobile devices and the peer-to-peer nature of the network lead to strong mutual dependencies among devices by requiring them to cooperate to achieve common goals. For example, in a planetary exploration application, miniature rovers, each equipped with a single sensor or actuator, may need to perform experiments that demand data from different combinations of sensors and actuators. The need for different devices to cooperate in a dynamic environment is the fundamental motivator of enhanced coordination support.

Mechanisms that address the complexities of ad hoc networks include enhancements to the operating system [105, 81], specialized languages [130, 46], and middleware [41, 29, 126, 42]. Operating systems are tightly integrated with low-level communication services, platform-dependent, and expose unnecessary details that complicate application programming tasks. The development and use of new programming languages is costly as it requires teaching developers a new language. In contrast, middleware simplifies application development by providing higher-level abstractions, while minimizing risk by building upon the existing software infrastructure and developer familiarity with existing programming languages. When designed properly, middleware can divert attention from mundane concerns like low-level protocol development, to more fruitful areas directly involving application-specific goals.

Designing a coordination middleware for ad hoc networks is difficult. On the one hand, the middleware must provide higher-level abstractions to simplify application development. On the other hand, the middleware must not place too many constraints on, or make too many assumptions about, the behavior of the underlying network, lest it stop working when deployed in a network that violates these assumptions. It must be flexible to adapt to a dynamic environment. For example, consider an application in which an universal remote held by the user interacts with the set of devices within wireless range. As the user moves, the middleware must adapt to changes in the set of devices within range. Furthermore, wireless signals are prone to transient interference due to contention with other devices and multi-path effects (i.e., Rayleigh fading [159]) from the environment. The middleware must be designed to handle unpredictable message loss and transient network connectivity among neighboring devices.

This chapter introduces Limone, a lightweight coordination model and middleware for highly dynamic MANETs. It supports logical mobility of software agents and physical mobility of devices. Limone agents are software processes that represent units of modularity, execution, and mobility. In a significant departure from other coordination frameworks, Limone emphasizes agent individuality by focusing on asymmetric interactions among agents. Asymmetry is good because it minimizes the overhead of the coordination middleware. For example, suppose agent A needs to communicate with agent B but not vice-versa. In this case, an asymmetric middleware can facilitate communication from A to B without having to enable communication from B to A. To achieve this asymmetry, the middleware maintains for each agent a

separate *acquaintance list* that defines a personalized view of remote agents within communication range. For each agent, Limone discovers remote agents and updates its acquaintance list according to customizable policies.

Traditional Linda-like tuple space primitives facilitate the coordination of agent activities. However, Limone allows each agent to maintain strict control over its local data, provides advanced pattern matching capabilities, permits agents to restrict the scope of their operations, and offers a powerful repertoire of reactive programming constructs. The autonomy of each agent is maintained by the exclusion of distributed transactions and remote blocking operations. Furthermore, Limone ensures that all distributed operations contain built-in mechanisms to prevent deadlock due to packet loss or disconnection. For these reasons, Limone is resilient to message loss and unexpected disconnection. This allows Limone to function in realistic and highly dynamic mobile ad hoc environments in which other middleware cannot.

The remainder of this section discusses an example application, `Universal Remote`, that is used as a running example throughout the rest of this chapter. In addition to discussing how `Universal Remote` can be implemented using Limone, several other coordination systems are also discussed, compared, and contrasted. By comparing and contrasting Limone's approach with that of alternative systems, a better understanding of the motivations behind Limone's design is attained.

3.1.1 Application Example: Universal Remote

Consider `Universal Remote`, an application in which a mobile device held by the user discovers controllable devices within wireless range, and enables the user to control them over a MANET. As the user moves, the set of devices within range vary, which must be reflected by changes presented to the user. The timing and locations in which the user moves are unknown and uncontrolled by the application. This illustrates significant challenges intrinsic to MANETs, like unpredictable and dynamic network connectivity. The remainder of this section discusses how Universal Remote can be implemented using Limone and alternative coordination models including JEDI [41], MARS [29], and LIME [126].

JEDI offers a publish-subscribe model where nodes interact by exchanging events through a logically centralized, though physically distributed, event dispatcher. An event is modeled as an ordered set of strings where the first is the name of the event, and the rest are application-specified parameters. Nodes subscribe to events using regular expressions on the event name. When a node publishes an event, the event dispatcher passes it to all nodes subscribed to it. Since all communication is done through the event dispatcher, publishers are decoupled from subscribers.

`Universal Remote` can be implemented in JEDI as follows. To discover devices, each devices can publish a *device description event* to which the user subscribes. When a device is no longer within range, the system can publish a special event to announce the disconnection. Similarly, to control a device, the user can publish *device control events* with instructions for a particular device. Each device subscribes to the control events that are destined for it. While this design works, it has a major drawback, which is the fact that events are not persistent. When a device publishes a description event, the event dispatcher immediately passes it to all user devices subscribed to it. Once the event is delivered to all known subscribers, the event is discarded. This is problematic because in a mobile environment, the user may not be present when the event is published. Thus each device must periodically re-publish its description event, which is inefficient in terms of energy and network bandwidth.

MARS consists of logically mobile agents that can migrate across devices in the network. Each device maintains a local tuple space that is accessable only to agents residing on the device. The tuple space is enhanced with reactions that allow an agent to respond to certain events like the insertion or removal of a tuple. MARS agents can only coordinate with co-located agents (i.e., agents residing on the same device). Agent migration is required for inter-device communication. MARS adapts to mobile environments by allowing agents to "catch" connection events that indicate the arrival or disconnection of a remote device.

`Universal Remote` can be implemented in MARS as follows. Whenever a device detects a user, it spawns an agent that migrates onto the user's device and inserts a *device description tuple*. The user's agent reacts to this tuple, thus learning about the device. A similar mechanism can be used by the user to control devices that are in range. This design is inefficient since it requires agent migration for each operation.

Like MARS, LIME provides a coordination model based on logically mobile agents that can migrate across physically mobile devices. Unlike MARS, LIME maintains group-level tuple spaces that span entire groups of devices. Each device maintains a host-level tuple space that is restricted to the local device. When multiple devices come within wireless range, they form a group and logically merge the contents of their individual host-level tuple spaces, creating a single logically-centralized group-level tuple space from the agent's (and application programmer's) perspective. Agents coordinate by exchanging tuples through the group-level tuple space. Tuple space reactions allow the system to notify an agent when a particular tuple is in the tuple space. LIME provides strong atomicity and functional guarantees via distributed transactions. For example, when two groups merge, the logical merging of the two group-level tuple spaces is done atomically. While powerful, this requires a symmetric relationship between devices, which increases overhead, and assumes connectivity throughout the transaction, which may be difficult to guarantee depending on the level of dynamics within the network.

`Universal Remote` can be implemented in LIME as follows. To enable device discovery, each device inserts *device description tuples* into the host-level tuple space, to which the user reacts. Likewise, devices are controlled by having the user insert *control tuples* into the tuple space to which the targeted device reacts. The main problem is the symmetric relationship between devices. LIME enforces a symmetric relationship between coordinating nodes by forming *groups*. All devices that are controllable by a universal remote must be part of the same group as the universal remote. The limitation is that a device can only be in one group at a time. That is, it can only be controlled by one universal remote at a time, limiting device-sharing and scalability.

In contrast, Limone follows an incremental and agent-centric paradigm where each agent forms it own group via the aforementioned acquaintance list. When a user's agent and a device's agent come within wireless range, the exchanging of tuples and reactions occur gradually, not in a single atomic operation. This is because ensuring atomicity may be impossible when the network is highly dynamic.

Key differences between LIME and Limone lie in the engagement policy and the number of tuple spaces used. LIME's engagement policy is symmetric and built into the

model whereas Limone's policy is customizable by the application, via the acquaintance policy, and asymmetric, meaning agent `B` may be in agent `A`'s acquaintance list, but not vice-versa. In Limone, each agent has a seperate tuple space, whereas in LIME all agents on a host share a single logically-centralized group-level tuple space.

Due to the reliance on lightweight and simple coordination constructs, Limone does not provide the same levels of atomicity guarantees as LIME. However, it can provide the general functionality of LIME's distributed operations with relaxed atomicity guarantees. For example, LIME provides a global **in** operation that atomically searches the entire group-level tuple space, which requires locking all hosts in the group simultaneously. By doing this, LIME guarantees that if a matching tuple exists at the time the operation is issued, the tuple will be found. Although Limone cannot provide such a guarantee, the user's agent can sequentially perform **inp** on each agent in the acquaintance list until it finds a match. While this does not guarantee the match will be found, the probability of success is high since the match will be found so long as it remains in the agent's tuple space, which is the case for the device description tuple. This reflects the highly pragmatic approach the design of Limone has followed.

The universal remote can be implemented in existing coordination models, but results in implementations that limit efficiency and flexibility. In the next section, we introduce a new coordination model called Limone that addresses the issues identified in this section.

3.2 Programming Model

Limone assumes a computational model consisting of mobile devices (hosts) capable of forming ad hoc networks; mobile agents that reside on hosts but can migrate from one host to another; and data owned by agents that is stored in local tuple spaces that belong to indvidual agents. The relationship between hosts, agents, and tuple spaces is shown in Figure 3.1. The features of Limone can be broadly divided into four general categories: context management, explicit data access, reactive programming, and code mobility.

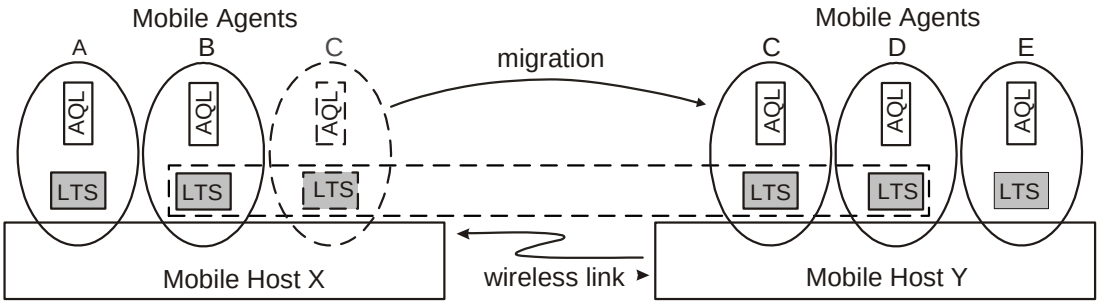


Figure 3.1: The Limone model. Software agents are represented by ovals. Each agent owns a local tuple space (LTS) and an acquaintance list (AQL). Agent C is shown migrating to host Y. The dotted rectangle surrounding the tuples spaces of agents B, C, and D highlight those that are accessible by agent C.

Context Management: The Acquaintance List

Central to context management is an agent's ability to discover neighboring agents and to selectively determine their relevance. Limone provides a beacon-based discovery protocol that informs each agent of the arrival and departure of other agents. Limone notifies each agent of its relevant neighbors by storing them in indivudal acquaintance lists, one per agent, where relevance is determined by an *engagement policy* that is agent-specified. Since each agent has different neighbors and engagement policies, the context of each agent may differ from that of its peers.

Many existing coordination models for mobility in ad hoc environments presume a symmetric and transitive coordination relationship among agents. That is, if agent `A` coordinates with agent `B`, then agent `B` must coordinate with agent `A`. The problem with such an approach is its limited scalability, since transitively applying the symmetric relationship results in the formation of large groups of agents in which every agent coordinates with every other agent in the group. As the number of agents increases, the likelihood that some move away also increases. This results in frequent group reconfigurations, which consume valuable resources. By allowing an agent to restrict coordination to agents it is interested in, and not enforcing a symmetric and transitive coordination relationship among groups of agents, Limone scales better while reducing resource utilization. For example, if an agent is surrounded by hundreds of agents but is interested only in two of them, it can concentrate on these two and ignore the rest. Concentrating on these two agents in no way impedes upon their

ability to coordinate with other agents. This asymmetry increases the level of decoupling among agents and results in a more robust coordination model that requires fewer assumptions about the underlying network [84].

Explicit Data Access: Localized Tuple Spaces

Limone accomplishes explicit data access in the following manner. Each agent owns a single tuple space that provides operations for inserting and retrieving tuples. Explicit data access spans at most two agents. The agent initiating the data access (called the reference agent) must have the destination agent in its acquaintance list. For security reasons, the semantics of the operation is akin to a *request*, i.e., the reference agent requests that the destination agent perform the tuple space operation for it. By doing this, each agent maintains full control over its local data and can implement policies for rejecting and accepting requests from remote agents. This is accomplished using an *operation manager*, which is described next.

The operation manager controls which requests are performed and is customizable on a per-agent basis. It greatly enhances the expressiveness of Limone since it can be customized to perform relatively complex tasks. For example, suppose each agent creates a public/private key pair and publishes its public key in a "read-only" tuple. The read-only nature of this tuple can be enforced by the operation manager by preventing all requests that would remove it from executing. Using this read-only tuple, secrecy and authentication can be achieved. Suppose a reference agent wishes to place a tuple onto a remote agent's tuple space. To do this, it can first encrypt the data using its private key, then by the remote agent's public key. The remote agent knows that the tuple is secret if it is able to decrypt it using its private key. It also knows that the tuple was sent by the reference agent if it can decrypt it using the reference agent's public key. This example illustrates how the operation manager can be configured to perform complex tasks, in this case authentication. Other tasks include access control, enforcing quality-of-service contracts, and fighting denial-of-service attacks by throttling response times prioritizing the tasks.

3.2.1 Reactive Programming: Tuple Space

Reactive programming constructs enable an agent to automatically respond to particular tuples in the tuple spaces belonging to agents in its acquaintance list. Two state variables within each agent, the *reaction registry* and *reaction list,* support this behavior. A reference agent registers a reaction by placing it in its reaction registry. Once registered, Limone automatically propagates the reaction to all agents in the acquaintance list that satisfy certain properties specified by the reaction (e.g., the agent's name or location). At the receiving end, the operation manager determines whether to accept the reaction. If accepted, the reaction is placed into the reaction list, which holds the reactions that apply to the local tuple space.

When the tuple space contains a tuple satisfying the trigger for a reaction in the reaction list, the agent that registered the reaction is sent a notification containing a copy of the tuple and a value identifying which reaction was fired. When this agent receives the notification, it executes the code associated with the reaction atomically. This mechanism, originally introduced in Mobile UNITY [153], and later deployed in LIME, is distinct from that employed in traditional publish/subscribe systems in that it reacts to state properties rather than to data operations. For instance, when a new agent is added to the acquaintance list, its tuples may trigger reactions regardless of whether the new agent performed any operations.

Code Mobility: Mobile Agents

Code mobility is supported in Limone by allowing agents to migrate from one host to another. When an agent migrates, Limone automatically updates its context and reactions. There are many benefits to agent to migration. For example, if a particular host has a large amount of data, an agent that needs access to it can relocate to the host holding the data and thus have reliable and efficient access to it despite frequent disconnection among hosts. Another example is software update deployment. Suppose an agent is performing a certain task and a developer creates a new agent that can perform the task more efficiently. The old agent can be designed to shutdown when the new agent arrives. Thus, simply having the new agent migrate

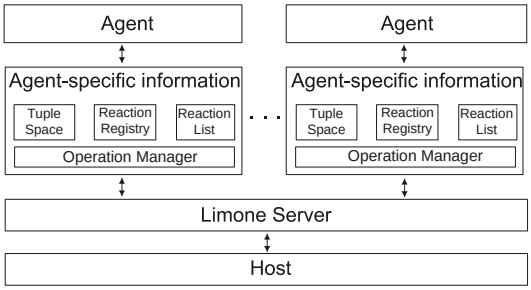


Figure 3.2: The overall structure of Limone.

to the same host as the old agent updates the application. Such updates are common practice on the Internet and are equally benificial in MANETs.

3.3 Middleware Implementation

Limone provides a runtime environment for agents via the Limone server, a software layer between the agent and the underlying operating system. By using different ports, multiple Limone servers may operate on a single host. However, for the sake of simplicity, we will treat each host as having one Limone server.

An application uses Limone by interacting with an agent that is specific to the application. Each agent contains a tuple space, acquaintance list, reaction registry, reaction list, and operation manager. The overall structure of Limone is shown in Figure 3.2. An agent allows the application to customize its profile, engagement policy, and operation manager. An agent's profile describes its properties. Its engagement policy specifies which agents are relevant based on their profiles. The operation manager specifies which remote operation requests are accepted. This section describes how Limone fulfills its responsibilities and is organized around the key elements of the run-time environment, i.e., agent discovery, tuple space management, reactions, and agent mobility.

Discovery Mechanism. Since connectivity between hosts in an ad hoc network is dynamic, Limone provides a *discovery protocol* based on beacons that allows an agent to discover the arrival and departure of other agents. Each Limone server periodically broadcasts a beacon containing a *profile* for each agent running on top of it. A profile

ABSTRACT STATE: A set of profiles, $\{p_1, p_2, \ldots\}$

INTERFACE SPECIFICATION:
boolean contains(AgentID aID) — Returns *true* if the list contains a profile that has the specified AgentID.
Profile[] getApplicableAgents(ProfileSelector[] pss) — Returns all of the profiles within the list that match any of the specified profile selectors.

Figure 3.3: Acquaintance list.

is a collection of triples each consisting of a property name, type, and value. The two system-defined entries include the host on which the agent resides and a unique agent identifier. Additional entries can be added by the application. When the Limone server receives a beacon, it forwards it to each of its agents. When an agent receives a beacon, it extracts the profiles and passes them to its *acquaintance handler,* which uses the agent's *engagement policy* to differentiate the relevant profiles and places them in the acquaintance list. If a particular agent's profile is already in the list, the acquaintance handler ensures that it is up to date and that it still satisfies the engagement policy. Once a profile is added to the acquaintance list, the acquaintance handler continuously monitors the beacons for the profile. If it is not received for an application-customizable period of time, the acquaintance handler removes the profile from the acquaintance list.

The acquaintance list, shown in Figure 3.3, contains a set of profiles representing the agents within range that have satisfied the engagement policy. The addition of a profile into the acquaintance list signifies an *engagement* between the reference agent and the agent represented by the profile. Once the reference agent has engaged with another agent, it gradually propagates its relevant reactive patterns (the trigger portion of the reaction) to the remote agent. While the addition of the profile to the acquaintance list is atomic, the propagation of reactive patterns is gradual, avoiding the need for a distributed transaction.

The removal of a remote agent's profile from the acquaintance list signifies *disengagement* between the reference and remote agent. This occurs when the remote agent moves out of radio range, as signified by the lack of beacon reception. When this occurs, the reference agent removes all of the remote agent's reactive patterns from its reaction list. The removal of the profile from the acquaintance list and the reactive

patterns from the reaction list is performed atomically, which is possible because it is done locally.

Tuple Space Management. All application data is stored in individually owned tuple spaces, each containing a set of tuples. Limone tuples contain data fields distinguished by name and store user-defined objects and their types. The ordered list of fields characterizing tuples in Linda is replaced by unordered collections of named fields in Limone. This results in a more expressive pattern matching mechanism similar to the Ψ-terms in [27] that can handle situations where a tuple's arity is not known in advance. For example, in the universal remote application, the following tuple may be created by the remote control destined for a device:

$$\begin{aligned} &\textit{tuple}\{\langle \text{“}\textit{type}\text{”},\ \textit{String},\ \text{“}\textit{command}\text{”}\rangle,\\ &\quad \langle \text{“}\textit{device ID}\text{”},\ \textit{String},\ \text{“}\textit{CD Player}\text{”}\rangle,\\ &\quad \langle \text{“}\textit{instruction}\text{”},\ \textit{String},\ \text{“}\textit{play}\text{”}\rangle\} \end{aligned}$$

Agents use templates to access tuples in the tuple space. A template is a collection of named constraints, each defining a name and a predicate called the *constraint function* that operates over the field type and value. A template matches a tuple if each constraint within the template has a matching field in the tuple. A constraint matches a field if the field's name, type, and value satisfy the constraint function. For example, the following template matches the message tuple given above:

$$\begin{aligned} &\textit{template}\{\langle \text{“}\textit{type}\text{”},\ \textit{String},\ \text{valEql(“}\textit{command}\text{”)}\rangle\}\\ &\quad \langle \text{“}\textit{device ID}\text{”},\ \textit{String},\ \text{valEql(“}\textit{CD Player}\text{”)}\rangle\}^{2} \end{aligned}$$

Notice that the tuple may contain more fields than the template has constraints. As long as each constraint in the template is satisfied by a field in the tuple, the tuple matches the template. This powerful style of pattern matching provides a higher degree of decoupling since it does not require prior knowledge of the ordering of fields within a tuple, or its arity, to create a template for it.

Local Tuple Space Operations. The operations an agent can perform on its tuple space are shown in Figure 3.4. The **out** operation places a tuple into the tuple space.

[2] `valEql(`p`)` is a constraint function that returns *true* if the value within the field is equal to p.

INTERFACE SPECIFICATION:
void out(Tuple t) — Places a tuple into the tuple space.
Tuple rd(Template template) — A blocking operation that returns a copy of a tuple matching the template.
Tuple rdp(Template template) — A non-blocking operation that returns a copy of a tuple matching the template, or ε if none exists.
Tuple[] rdg(Template template) — Same as **rd** except it returns a copy of all matching tuples.
Tuple[] rdgp(Template template) — Same as **rdp** except it returns all matching tuples that exist.
Tuple in(Template template) — Same as **rd** except it removes the tuple.
Tuple inp(Template template) — Same as **rdp** except it removes the tuple.
Tuple[] ing(Template template) — Same as **rdg** except it removes the tuples.
Tuple[] ingp(Template template) — Same as **ingp** except it removes the tuples.

Figure 3.4: Local tuple space operations.

The operations **in** and **rd** block until a tuple matching the template is found in the tuple space. When this occurs, **in** removes and returns the tuple, while **rd** returns a copy without removing it. The operations **inp** and **rdp** are the same as **in** and **rd** except they do not block. If no matching tuple exists within the tuple space, ε is returned. The operations **ing** and **rdg** are similar to **in** and **rd** except they find and return *all* matching tuples within the tuple space. Similarly, **ingp** and **rdgp** are identical to **ing** and **rdg** except they do not block. If they do not find a matching tuple, ε is returned. All of these operations are performed atomically, which is possible because they are performed locally.

Remote Tuple Space Operations. To allow for inter-agent coordination, an agent can request a remote agent to perform an operation on its tuple space. To do this, Limone provides remote operations **out**, **inp**, **rdp**, **ingp**, and **rdgp**, as shown in Figure 3.5. These differ from the local operations in that they require an `AgentLocation` parameter that specifies the target agent. When a remote operation is executed, the reference agent sends a request to the remote agent specified by the `AgentLocation`, sets a timer, and remains blocked till a response is received or the timer times out. When the remote agent receives the request, it passes it to its operation manager, which may reject or approve it. If rejected, an exception is returned to allow the reference agent to distinguish between a rejection and a communication failure. If

INTERFACE SPECIFICATION:

void out(AgentLocation loc, Tuple t) — Asks the agent at **loc** to place a tuple in its tuple space.

Tuple rdp(AgentLocation loc, Template template) — Asks the agent at **loc** to perform a **rdp** operation. Returns the results, or ε if the operation times out.

Tuple[] rdgp(AgentLocation loc, Template template) — Asks the agent at **loc** to perform a **rdgp** operation. Returns the results, or ε if the operation times out.

Tuple inp(AgentLocation loc, Template template) — Asks the agent at **loc** to perform a **inp** operation. Returns the results, or ε if the operation times out.

Tuple[] ingp(AgentLocation loc, Template template) — Asks the agent at **loc** to perform a **ingp** operation. Returns the results, or ε if the operation times out.

Figure 3.5: Operations on a remote tuple space.

accepted, the operation is performed atomically on the remote agent, and the results are sent back. The timer is necessary to prevent deadlock due to message loss. If the request or response is lost, the operation will time-out and return ε. To resolve the case when an operation times out while the response is still in transit, each request is enumerated and the remote agent includes this value in its response.

Reaction Mechanism. Reactions enable an agent to inform other agents within its acquaintance list of its interest in tuples matching a particular template. A reaction contains an application-defined call-back function that is executed by the agent that created it when a matching tuple is found in a tuple space that the reaction is registered on. Reactions fit particularly well with ad hoc networks because they provide an asynchronous form of communication between agents by transferring the responsibility of searching for a tuple from one agent to another, which eliminates the need to continuously poll for data.

A reaction consists of a *reactive pattern* and a *call-back function*. The reactive pattern contains a template that indicates which tuples trigger it and a list of profile selectors that determine which agents the reaction should propagate to. The call-back function executes when the reaction *fires* in response to the presence of a tuple that matches its template in the tuple space it is registered on. The firing of a reaction consists of sending a copy of the matching tuple to the agent that registered the reaction. When the matching tuple is received, the reference agent executes the reaction's call-back function atomically. To prevent deadlock, the call-back function cannot

ABSTRACT STATE: — A set of reactions, $\{r, \ldots\}$

INTERFACE SPECIFICATION:

ReactionID addReaction(Reaction rxn)— Adds a reaction to the reaction registry and returns the reaction's ReactionID.

Reaction removeReaction(ReactionID rID) — Removes and returns the reaction with the specified ReactionID.

Reaction get(ReactionID rID) — Retrieves the reaction with the specified ReactionID.

Reaction get(Profile profile) — Retrieves all reactions containing profile selectors that match the given profile.

Figure 3.6: Reaction Registry.

perform blocking operations. Notice that the message containing the tuple may be lost, meaning there is no guarantee that a reaction's callback function will be executed even if a matching tuple is found.

The list of profile selectors within the reactive pattern determines which agents it should be propagated to. Implementation-wise, a profile selector is a template and a profile is a tuple. They are subject to the same pattern matching mechanism but are functionally different because profiles are not placed in tuple spaces. A reaction's reactive pattern propagates to a remote agent if the remote agent's profile matches *any* of the reactive pattern's profile selectors. Multiple profile selectors are used to lend the developer greater flexibility in specifying a reaction's domain. For example, returning to our universal remote example, a device would have the following profile:

$$\textit{profile}\{\langle \text{“}\textit{type}\text{”},\ \textit{String},\ \text{“}\textit{Device}\text{”}\rangle\}$$

and a reaction created by the universal remote control would contain the following profile selector to restrict its propagation to device agents:

$$\textit{profile selector}\{\langle \text{“}\textit{type}\text{”},\ \textit{String},\ \texttt{valEql}(\text{“}\textit{Device}\text{”})\rangle\}$$

This ensures that the reactive pattern only propagates to agents whose profile contains a property called "*type*," with a *String* value equal to "*Device*."

As in LIME, reactions may be of two types: ONCE or ONCE_PER_TUPLE. The type of the reaction determines how long it remains active once registered on a tuple space. A ONCE reaction fires once and automatically deregisters itself after firing. When a ONCE reaction fires and the agent that owns the reaction receives the resulting tuple(s), it deregisters the reaction from all other agents, preventing the reaction from firing later. If a ONCE reaction fires several times simultaneously on different tuple spaces, the reference agent chooses one of the results non-deterministically and discards the rest. This does not result in data loss because no tuples were removed. ONCE_PER_TUPLE reactions remain registered after firing, thus firing once for each matching tuple. These reactions are deregistered at the agent's request or when network connectivity to the agent is lost. To keep Limone as lightweight as possible, no history is maintained regarding where reactions were registered. Thus, if network connectivity breaks and later reforms, the formerly registered reactions will be re-registered and will fire again.

Two additional state components, the *reaction registry* and *reaction list,* are required for the reaction mechanism. The reaction registry, shown in Figure 3.6, holds all reactions created and registered by the reference agent. An agent uses its reaction registry to determine which reactions should be propagated following an engagement and to obtain a reaction's call-back function when a reaction fires.

The reaction list, shown in Figure 3.7, contains the reactive patterns registered on the reference agent's tuple space. The reactive patterns within this list may come from *any* agent within communication range, including agents *not* in the acquaintance list. Thus, to maintain the validity of the reaction list, the acquaintance handler notifies its agent when *any* agent moves out of communication range, not just the agents within its acquaintance list. The reaction list determines which reactions should fire when a tuple is placed into the local tuple space or when a reactive pattern is added to it.

Agent Mobility. Coordination within Limone is based on the logical mobility of agents and physical mobility of hosts. Agents are logically mobile since they can migrate from one host to another. Agent mobility is accomplished using μCode [142]. μCode provides primitives that support light-weight mobility preserving code and state. Of particular interest is the μCodeServer and μAgent. A μAgent maintains

ABSTRACT STATE: — A set of reactive patterns, $\{rp_1, rp_2, \ldots\}$

INTERFACE SPECIFICATION:
boolean addReactivePattern(ReactivePattern rp) — Adds a reactive pattern to the reaction list, returns true if it was successfully added.
void clear() — Clears the reaction list by removing all reactive patterns within it.
ReactivePattern[] getApplicablePatterns(Tuple tuple) — Retrieves all of the reactive patterns within the list that should fire on the specified tuple.
void removeReactivePattern(ReactivePattern rp) — Removes the specified reactive pattern from the list if it is in the list.
void removeReactivePatterns(AgentID aID) — Removes all reactive patterns from the list that were registered by the agent with the specified AgentID.

Figure 3.7: Reaction List.

a reference to a μCodeServer and provides a `go(String` *destination*`)` method that moves the agent's code and data state to the destination. The agent's thread state is not preserved because doing so would require modifying the Java virtual machine, limiting Limone to proprietary interpreters. Thus, after an agent migrates to a new host, it will start fresh with its variables initialized to the values they were prior to migration.

Limone cooperates with μCode by running a μCodeServer alongside each Limone server and having the Limone agent extend μAgent. By extending μAgent, the Limone agent inherits the `go(String` *destination*`)` method. However, Limone abstracts this into a `migrate(HostID` hID`)` method that moves the agent to the destination host by translating the `HostID` to the string accepted by μCode. Prior to migration, the agent first deregisters all of its reactive patterns from remote agents, and removes its profile from the beacons. Once on the new host, the agent resumes the broadcasting of its beacons. This allows remote agents to re-engage with the agent at its new location.

3.4 Microbenchmarks

A prototype of Limone has been implemented in Java. The implementation adheres to the model given in Section 3.2, where each construct is a distinct object that

Model	Lines of Code	Time (ms)
Limone	250	50.3
LIME	170	73.6
Raw Sockets	695	44.6

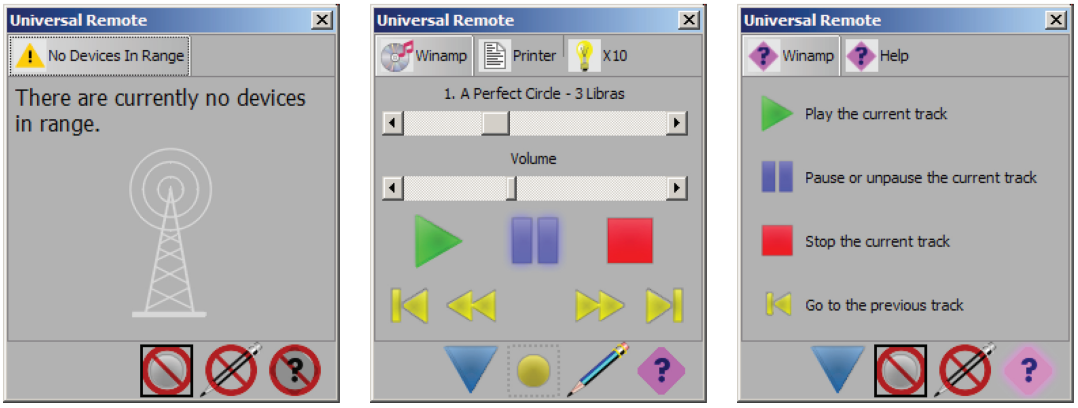
Figure 3.8: Application code size and round-trip message passing time using reactions as a trigger, averaged over 100 rounds.

implements the interface and behavior described in Section 3.3. A `LimoneServer` object serves as a foundation that listens for incoming messages and beacons. It periodically broadcasts beacons containing the profiles of all agents residing on it. An application can load its agents onto the `LimoneServer` by calling `loadAgent()`, or by using a special `Launcher` object that communicates with the server through its single-cast port. This allows new agents to be dynamically loaded, possibly from a remote device.

As a testament to how lightweight Limone is, its jar file is only 111.7KB. To analyze its performance, the round trip time for a tuple containing eight bytes of data to be pulled onto a remote agent and back using reactions as triggers is calculated. The test was performed using two 750MHz laptops running Java 1.4.1 in 802.11b ad hoc mode with a one second beaconing period. The laptops were located in a "clean room" environment where they are stationary and placed next to each other. To compare Limone's performance, we also performed the same operation using LIME and raw TCP sockets. Averaged over 100 rounds, the results of our tests are shown in Figure 3.8. They show that Limone adds some overhead over raw sockets, but not as much as LIME. In this simple test, Limone requires more code than LIME because of Limone's more expressive pattern matching mechanism and engagement policy.

3.5 Application Case Study: Universal Remote

This section presents an application developed using Limone. The application is called *Universal Remote*. It consists of a remote control held by the user that automatically discovers devices within range and enables the user to control them. Limone is ideal for this application because it automatically discovers all controllable devices within



(a) Before finding devices. (b) After finding devices. (c) Displaying device help.

Figure 3.9: The Universal Remote Application's User Interface

range, allows the device's state to be shared and controlled by multiple remotes, and is lightweight enough to run on embedded devices like electrical appliances.

When the Universal Remote is started, it displays the notice shown in Figure 3.9(a) while it finds devices in range. When it does, it displays them in a tabbed list shown in Figure 3.9(b). Devices that come into range are added to the list, while devices that go out of range are removed. This ensures that users cannot control devices that are no longer available. In addition to the controls for each device, a fixed row of controls along the bottom of the window enable the user to customize the display and view the context-sensitive help, as shown in Figure 3.9(c).

Each device runs a `LimoneServer` and has an associated agent. These agents insert information about the device into their local tuple space: namely, the advertised list of controls (buttons, sliders, etc.), the "help text" associated with each of these controls, and the current state of each control.

In order to further simplify the creation of device agents, we implemented a `GenericDeviceAgent` class as well as a `DeviceDefinition` interface. The `GenericDeviceAgent` is a Limone agent that accepts any `DeviceDefinition` interface as a plug-in; this interface exposes information about the device (such as its advertised controls) to the `GenericDeviceAgent` as well as exposing specific operations (i.e., pressed buttons or moved sliders) to the device. This allows the device-specific code to be implemented with little to no knowledge of Limone.

As an example, we simulated a remotely-controllable stereo by writing a `WinampDeviceDef` to control Winamp [179]. This required implementing the eleven methods in the `DeviceDefinition` interface, which took about 250 lines of code and about an hour to write. The agent is started by loading a `GenericDeviceAgent` onto the device running Winamp, and instructing it to interface with the `WinampDeviceDef` class.

The `GenericDeviceAgent` instantiates a `WinampDeviceDef` and obtains basic information about the device such as its name, icon, functions, state, and help text. Based on this information, the `GenericDeviceAgent` inserts advertisement, state, and help text tuples into its tuple space using the local **out** operation, which the `UniversalRemoteAgent` reacts to and uses to create its display.

When the `UniversalRemoteAgent` alters the state of a device (such as by toggling a button), it creates an `ActionTuple` that describes the change and inserts it into the device's local tuple space using the remote **out** operation. The `GenericDeviceAgent` reacts to this tuple and notifies the `WinampDeviceDef`, which handles the change (such as by pausing the song if the pause button was toggled) and passes any change to the device's state (such as that the pause button is now lit) back to the `GenericDeviceAgent`. The `GenericDeviceAgent` then encapsulates the information in a `StateTuple` and inserts it into its local tuple space using the local **out** operation. The `UniversalRemoteAgent` reacts to this `StateTuple` and updates its display accordingly.

Aside from the SWT graphics library, no third-party libraries were needed in the implementation of the `UniversalRemoteAgent`, and no third-party libraries were needed for the implementation of the device agents aside from libraries specific to each device (e.g., an X10 communication library for the X10 agent). Further, since Limone uses a small subset of the Java API, both the client and server could be run on a device with limited Java support, like a PocketPC.

3.6 Chapter Summary

Limone is a lightweight but highly expressive coordination model and middleware tailored to meet the needs of developers concerned with mobile applications over ad hoc networks. Central to Limone is the management of context-awareness in a highly dynamic setting. The context is managed transparently and is subject to policies imposed by each agent in response to its own needs at a particular point in time. Explicit manipulation of the context is provided by operations that access data owned by agents in the acquaintance list. Each agent retains full control of its local tuple space since all remote operations are simply requests to perform a particular operation for a remote agent and are subject to policies specified by the operation manager. This security provision encourages a collaborative type of interaction among agents. An innovative adaptation of the reaction construct facilitates rapid response to environmental changes. As supported by evidence to date, the result of this unique combination of context management features is a coordination model and middleware that promise to reduce development time for mobile applications.

Chapter 4

Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks

Agilla is the first mobile agent middleware to be successfully implemented and evaluated on devices representative of those found in most modern WSNs. It differs from Limone by focusing on the WSN platform, where devices are embedded and have less computation, energy, and communication resources by several orders of magnitude. The limited resources available in WSNs results in Agilla sharing Limone's objective of being as lightweight as possible. Designed to support self-adaptive applications, Agilla provides a programming model in which applications consist of evolving communities of agents that share a WSN and coordinate through device-level tuple spaces. Agents can dynamically enter and exit a network and can autonomously clone and migrate themselves in response to changes in the environment. Prior to Agilla, WSNs usually ran statically-installed software, limiting their flexibility. Using Agilla, a WSN is deployed with no pre-installed application. Instead, users inject mobile agents that spread across nodes performing application-specific tasks. Each agent is autonomous, allowing multiple applications to share a network. Implemented on top of TinyOS [73], a popular event-based operating system used in WSNs, Agilla's feasibility and efficiency was demonstrated by experimental evaluation on two physical testbeds consisting of Mica2 and TelosB WSN devices. In addition, Agilla's ability to support self-adaptive WSN applications has been demonstrated in the context of several applications including fire detection and tracking, monitoring cargo containers, and robot navigation.

4.1 Motivation

Prior to discussing the details motivating Agilla, first consider the following example scenario demonstrating the need for a framework supporting self-adaptive applications.

> In the remote arid forests of central Arizona, an ember glows in a fire pit left by a careless hiker. A slight wind gives new life to the embers while pushing dry leaves into the pit. The leaves start to burn and the fire grows. After months of drought and years of accumulating underbrush, the fire finds plenty of fuel and quickly spreads with the prevailing winds. The remoteness of the region would allow the fire to burn undetected for hours, virtually ensuring that it will soon rage out of control eventually turning into a giant fire storm. Fortunately, the USDA Forest Service had recognized this area as highly incendiary and deployed a WSN for detecting fire. As the fire grows, nearby sensors quickly detect it and spawn tracking agents that swarm around the fire, collecting information about the exact location of the flames. The tracking agents form a dynamic perimeter jumping away from the fire as it draws too near, and cloning themselves onto neighbors to encompass the growing fire. Simultaneously, they notify a base station that forwards the warning via the Internet to the nearest fire fighters a hundred miles away. By the time they arrive, the entire region is engulfed, burning with such intensity so as to be seen and felt from miles away.
>
> The fire fighters act quickly and make it their first priority to evacuate the area. They inject search-and-rescue agents into the network that spread and repeatedly clone themselves, scouring the region looking for lost hikers trapped by the flames. Some of these agents find a group of children and coordinate with the other agents to form a path of greatest safety that the rescuers, carrying PDAs to access the path information, use to reach the children and bring them to safety. Once everyone is safe, the fire fighters query the tracking agents for the precise location and dynamics of the fire. From this data, they are able to predict the fire's behavior and strategically control its movements, preventing it from

approaching populated areas where property can be damaged and people injured.

As the fire dies and shifts positions, additional sensors are dropped from airplanes in the regions previously engulfed to make up for lost nodes. Within seconds, the existing agents discover the new nodes and clone themselves onto them. Once the fire has died and the network has been replenished, the application enters a low-overhead state where the tracking agents also die leaving only small fire detection agents that periodically search for fire. The minuscule amount of resources consumed by these agents allow other applications to run, which biologists take advantage of by injecting state-of-the-art habitat monitoring agents for learning about the life cycle of coyotes.

As clearly shown by the scenario just presented, WSN applications must adapt to changes in the physical environment because WSNs are embedded in and responsible for monitoring a highly dynamic world. In the aforementioned scenario, the WSN responsible for tracking wildfire must adapt to changes in the wildfire's position. Another example is a WSN that provides a robot with extended situational awareness. In this case, it must adapt to the movements of the robot. New programming models and middleware are needed to support application self-adaptability within WSNs.

To address this need, ***Agilla*** is a middleware specifically designed to support self-adaptive applications in WSNs. Agilla structures an application as one or more *mobile agents*, which are special processes that can explicitly migrate or clone from node to node while maintaining their state. To ensure that agents remain autonomous while enabling inter-agent coordination, Agilla provides localized device-level tuple spaces that are remotely accessible. This allows agents to freely migrate while remaining able to coordinate with other agents.

Agilla's programming model, unique to those targeting WSNs, enables higher degrees of application self-adaptability. By facilitating agent migration, an application can restrict itself to reside only on relevant nodes. As the environment changes, the application can self-adapt by migrating its agents to positions that best fulfill its goals. Moreover, since tuple spaces facilitate spatiotemporal decoupling between agents, each agent can be replaced without affecting the other agents in the network. This

allows the set of mobile agents belonging to an application to evolve, enabling the application to adapt to changing requirements.

This chapter presents Agilla's programming model and the various engineering decisions that tailor Agilla to the WSN environment. Specifically, it makes the following primary contributions.

- Agilla enables for the first time the use of mobile agents and tuple spaces as fundamental programming models for self-adaptive applications in WSNs.
- Agilla includes the first implementation of a system supporting mobile agents and inter-agent coordination in highly resource-constrained wireless sensor platforms. Agilla's middleware implementation consumes only 57KB of code and 3.3KB of data memory on Mica2 platforms, and 45KB of code and 3.4KB of data memory on TelosB platforms.
- The efficiency of Agilla is evaluated via experiments on two physical WSN testbeds. The results show that Agilla agents can reliably migrate one hop every 0.3 seconds and perform remote tuple space operations within 55ms on the Mica2 platform, which is sufficiently fast for many WSN applications.
- The generality and efficacy of the Agilla programming model for self-adaptive applications is demonstrated via three application case studies. They include wildfire tracking, cargo tracking, and robot navigation.

Mobile agents have previously been considered for use in WSNs. For example, mobile agents can perform certain operations like data integration [148, 149, 150] and tracking [167] better than traditional client/server message-passing mechanisms, and can be made energy efficient [117, 166]. While these efforts promote the use of mobile agents in WSNs, they were not actually deployed or evaluated in a real WSN. Instead, they were evaluated theoretically, in simulation, or using networks consisting of relatively resource-rich devices more akin to those found in MANETs.

Agilla is the first system to bring the mobile agent programming model into a real WSN. By integrating the mobile agent and tuple space programming models, Agilla

enables applications to be locally and autonomously self-adaptive. The Agilla programming model and middleware architecture meet the challenges unique to WSNs, e.g., severe resource constraints and unreliable wireless connectivity. The novel specialization of the mobile agent and tuple space programming models combined with a careful engineering effort resulted in the working Agilla middleware system described in this chapter.

4.1.1 Adaptation through In-network Reprogramming

As previously mentioned, a primary objective of Agilla is to facilitate the creation of adaptive WSN applications. One traditional approach for WSN adaption is to reprogram it over the wireless network. Systems that enable this can be divided based on what is reprogramed, i.e., native code, interpreted code, or both. Two systems that reprogram native code are Deluge [77] and MOAP [163]. They are designed to transfer large program binaries, enable the network to be arbitrarily reprogrammed, but incur high overhead and latency. To address this, SOS [68], Contiki [48], and Impala [107] are systems that enable partial reprogramming of binary code by providing a microkernel that supports dynamically linked modules. Since modules are relatively small, the cost of reprogramming is lower. Other systems that reprogram native code limit overhead by only sending the changes as determined by the diff [151] and rsync [82] algorithms.

Systems that reprogram interpreted code include Maté [98], application-specific virtual machines (ASVM) [99], Melete [185], and SensorWare [25]. In Maté and its successor, ASVM, applications are divided into capsules that are flooded throughout the network. Each node stores the most recent version of a capsule and runs the application by interpreting the capsules using a virtual machine (VM). Since capsules are flooded throughout the network, Maté and ASVMs are installed network-wide. Melete improves upon Maté by enabling multiple applications to co-exist within a sensor network. It does this by providing groups of capsules in which each group contains a different application. SensorWare allows users to dynamically inject mobile scripts into the network, enabling multiple applications to run concurrently. SensorWare scripts only provide weak mobility, i.e., they must restart after each migration, and the system was implemented for the relatively powerful iPAQ 3670 platform.

Strong mobility, in which a script maintains execution state across nodes, is useful in simplifying application code when the application requires computations to span multiple devices.

Hybrid systems that reprogram both native and interpreted code include VM* [90] and dynamic virtual machine (DVM) [13]. Both systems allow the instruction set to change, but do not focus on how applications adapt.

The aforementioned reprogramming systems share a common feature: the decision on when and where to reprogram the network is determined centrally at a base station, often by a human operator. In contrast, Agilla provides a fundamentally different programming model based on mobile agents and tuple spaces that are especially well-suited for self-adaptive applications in WSNs. Mobile agents can make adaptation decisions locally and autonomously *within* the network via migration (i.e., moving and cloning). Since network nodes are directly exposed to the environment, they can more quickly detect changes and better determine when software adaptation is necessary.

The remainder of the chapter is organized as follows. Section 4.2 presents Agilla's model and explains how it is tailored to WSNs. Section 4.3 discusses the various engineering tradeoffs necessary to cope with limited resources and an unreliable network. Section 4.4 presents the experimental results on Agilla's performance in terms of micro and macro-benchmarks. Section 4.5 discusses the use of Agilla to implement a fire detection and tracking application. Section 4.6 discusses how Agilla is used to guide a robot around dangerous areas by augmenting its onboard sensors. Section 4.7 discusses how Agilla is integrated with Limone to create a platform in which mobile agents can traverse multiple WSNs. Finally, a chapter summary is given in Section 4.8.

4.2 Programming Model

Agilla's model, shown in Figure 4.1, is designed to facilitate adaptive behavior within a WSN. Each node supports multiple autonomous *mobile agents* that can move or clone across nodes while carrying their state. Mobile agent interactions are facilitated

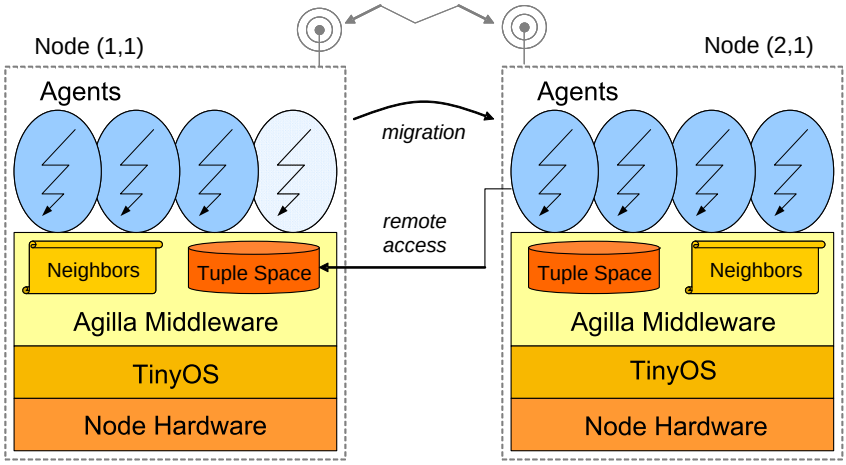


Figure 4.1: The Agilla model. Each node in the network maintains a node neighbor list, a discrete local tuple space, and multiple mobile agents. The mobile agents are able to migrate between nodes, and access the tuple spaces belonging to remote nodes.

by two basic data abstractions on each node: a *neighbor list* and a *tuple space*. Agilla provides support for both *local* and *remote* tuple space operations. Furthermore, it provides specialized *reaction* primitives that enable agents to efficiently respond to changing state. Since the spatial orientations of WSN nodes are significant, Agilla addresses them by their *location*, which allows applications to focus on their objective (e.g., sense a phenomena at a particular location). Each of these features are described later in this section.

Agilla's model is designed for *localized* adaptation like fire tracking. It is *not* meant for data collection applications that require deployment across the entire WSN. Agilla is especially suitable for handling situations in which local decisions would significantly reduce the amount of data wirelessly transmitted. For example, in a fire tracking application, allowing mobile agents to autonomously and locally adapt to the changing location of the fire prevents having to coordinate the application from a base station. At the same time, it prevents the application from needing to be deployed on every node in the network.

4.2.1 Mobile Agents

Mobile agents are special processes that can autonomously migrate across nodes. Agilla provides two forms of migration, strong and weak, to support diverse application needs for self-adaptation. Strong migration transfers both the code and state, allowing the agent to resume execution at the destination. It is useful for performing computations that span multiple nodes. Weak migration only migrates the code. It exhibits less overhead since the state does not need to be transferred, but resets the agent.

When an agent migrates, it can either clone or move. If an agent is cloned, a copy of it arrives and starts executing at the destination while the original one resumes on the original node. If an agent is moved, it will no longer exist on the original node after it arrives at the destination.

An agent's life cycle begins when it is either injected into the network from a base station, or cloned from another agent already in the network. Each agent executes autonomously, performing application-specific tasks, and multiple agents may reside on the same node. When an agent completes its tasks, it dies, freeing the computational resources it used.

4.2.2 Tuple Space

Agilla provides a *tuple space* [57] on each node, as shown in Figure 4.1. A tuple space is a type of shared memory in which data is structured as tuples that are accessed via pattern-matching. This enables a decoupled style of communication in which the sender and receiver need not agree on a shared memory address, or even coexist, for communication to occur. Using tuple spaces, agents can function autonomously and migrate freely while still being able to communicate. It is further motivated by the fact that WSN applications continuously evolve and the agents may change over time, meaning a particular agent may not know with which other agents it must communicate.

A tuple space in Agilla does not span multiple nodes to avoid the overhead of keeping it consistent in a dynamic environment and to ensure scalability. If a shared data abstraction spanning multiple nodes is necessary, it must be built on top of Agilla's primitives. Although a tuple space is contained on a single node, it can be accessed by agents locally and remotely, and it is augmented with reactions that enable agents to efficiently respond to changes in the tuple space state. Agilla tuple spaces also provide a convenient way for agents to discover properties of their environment. This is necessary because agents move and will encounter unfamiliar environments.

Agilla tuple spaces are accessible to agents residing on the same node via *local operations*, and to agents residing on different nodes using *remote operations*.

Local operations. An agent can save a tuple in the tuple space using the operation `out`, and can either remove a tuple using the operation `in` or read a tuple using operation `rd` (tuple space operations are named relative to the agent). The last two operations are blocking; they will wait until a matching tuple appears before allowing the agent to continue executing. Sometimes an agent may want to simply check whether a certain tuple exists and not block. To enable this, Agilla also provides *probing* operations `inp` (probing `in`), and `rdp` (probing `rd`). These operations are identical to `in` and `rd` except instead of blocking, they return `null` if a matching tuple does not exist.

All tuple space operations are performed atomically. This is feasible because a tuple space resides on a single node. A queue is used to serialize the operations, ensuring atomicity. If an operation blocks, it is placed in a separate wait queue to allow other operations to execute. When a tuple is inserted into the tuple space, the operations in the wait queue are placed in the regular queue, enabling them to search for a match. Thus, blocking operations are atomically executed at the time a matching tuple is found.

Remote operations. Agilla provides *remote tuple space operations* to enable agents residing on different nodes to communicate. They include remote versions of all non-blocking local operations, plus two special group operations, `routg` (remote group `out`) and `rrdpg` (remote probing group `rd`), that operate on all neighboring nodes. Remote operations are non-blocking to prevent an agent from deadlocking due to message loss.

Most remote tuple space operations rely on unicast communication. These operations are highly efficient since they only entail one network round trip, a request and a reply. If an underlying multi-hop networking service is available, these operations may be performed on tuple spaces residing on nodes multiple hops away. The only exceptions are the group operations, which use single-hop broadcast. This is feasible since wireless is a broadcast medium. Group operations are performed on a best-effort basis.

Reactions. Reactions provide interrupt semantics and consist of a template and a *call-back function*, which is executed when the reaction *fires*. Prior to firing, a reaction must first be *activated* by a matching tuple in the local tuple space. When a reaction fires, a copy of the matching tuple is given to the agent, and the agent executes the reaction's call-back function. Reactions allow an agent to indicate its interest in tuples that match a particular template. They persist across agent migrations, but do not maintain history across migrations. Thus, if an agent migrates away and then back, it will re-react to all matching tuples in the local tuple space. From experience developing applications on top of Agilla, this occurrence of re-reacting to the same tuples is either useful or rare depending on the purpose of the reaction. For example, in the fire tracking application, a reaction that is used to detect the presence of fire should re-react each time the agent moves within range of the fire. However, if the reaction is used for communication purposes, like notifying a tracking agent of where the fire is located, the tuple itself is removed during the execution of the reaction's call-back function, preventing the reaction from re-reacting to the same tuple.

Agilla reactions can only react to tuples in the *local* tuple space. In addition, to conserve memory, the call-back functions are *not* executed atomically relative to activation. In Agilla, reactions *eventually* fire as long as the matching tuples remain within a tuple space. While it is possible for a tuple to be inserted and removed without a reaction firing, adopting these weaker semantics reduces middleware complexity and overhead, which is necessary given the limited resources on a WSN node.

4.2.3 Location-Based Addressing

The spatial orientation of WSN nodes is important because they are embedded within the environment. To capture this, Agilla enables nodes to be addressed by their location, assuming such information is available. Thus, instead of performing a `rout` on node 1, an agent performs it on a node at location (x, y). This simplifies programming by allowing the developer to focus on what an agent does (e.g., measure the temperature at location (x, y)) rather than how it does it (e.g., perform a search for the node at location (x, y), then measure the temperature at that node). The location coordinate system must ensure that every node has a unique address. If there is no node present at a particular location, an acceptable error bound must be specified. Agilla primitives can be generalized to enable operations on a region. For example, a fire detection node may need to clone itself onto *all* or *at least one* node in a particular geographic area.

The determination and selection of a destination location is application-specific and must be specified by the application developer. For example, in an application that detects forest fires, the agents may perform a random walk. In another application involving a robot using mobile agents to discover its surroundings, the mobile agents may be restricted to the region surrounding the robot. The decision on how agents migrate impacts performance and should be a design criteria when developing an application. As a generic framework for developing WSN applications, Agilla does not control application-specific behavior like agent migration patterns.

The use of location-based addressing does not conflict with data-centric WSN applications [79]. Data centric applications route based on content. The same behavior can be achieved using agents and tuple spaces. Specifically, agents can be programmed to analyze a tuple and send the tuple to a particular neighbor based on its content. Agilla provides location-based addressing because agents control their own movements and the most natural and meaningful way to specify a destination is by location.

Location information can be obtained using GPS or various in-door localization systems [155, 76, 70, 164]. Obtaining location information does incur costs. If the cost is too high, Servilla application can optionally address nodes by their network address.

```
1: BEGIN  pushn fir
2:        pusht LOCATION
3:        pushc 2       // tuple <'fir', type:location> on stack
4:        pushc FIRE    // push reaction callback address
5:        regrxn        // register fire alert reaction
6:        wait          // wait for reaction to fire
7: FIRE   pop
8:        sclone        // strong clone to the node that detected the fire
9:        ...           // fire tracking code
```

Figure 4.2: A portion of the **FireTracker** agent

4.2.4 Example

Figure 4.2 shows a portion of a FireTracker agent that notifies it of a fire's location. To reduce code size, Agilla agents utilize a stack architecture. Thus, the various push instructions shown on lines 1-4 place items on the stack. The agent first registers a reaction sensitive to tuples containing the string "fir" and a location, and waits for the reaction to fire (lines 1-6). When a fire is detected, a matching tuple is inserted into the tuple space, causing the agent to execute the code beginning on line 7. Using the location stored within the tuple, the agent clones itself onto the node that detected the fire (line 8), and proceeds to form a dynamic perimeter around the fire. Note that FireTracker agents only reside on the nodes that are within the fire's vicinity, minimizing resource utilization. This differs from traditional deployments in which the application is installed onto every node in the network.

While Agilla agents are written in an assembly-like language, developing a higher-level language is a relatively straightforward extension of existing WSN scripting languages like TinyScript [97]. For example, TinyScript supports functions, which is a convenient way to access Agilla operations. Thus, instead of writing the instructions shown on lines 1-5 of Figure 4.2 that register a reaction, a high-level language modeled after TinyScript could execute a function like `registerReaction(FIRE,` $\langle$"fir", TYPE:LOCATION$\rangle$`)`. The reason why a higher-level language is not provided by Agilla is because the main contribution of Agilla is not the programming language, but rather the programming model.

4.2.5 Scalability

Wireless sensor networks have the potential to grow in size [128], making scalability an important attribute to consider. In fact, one reason Agilla maintains a separate tuple space on each node is to ensure scalability. While scalability ultimately depends on the way an application is designed, Agilla encourages scalable designs by promoting localized interactions between agents via single-hop neighbor lists on each node. Using this neighbor list and per-node tuple spaces, application algorithms must be designed to only use local knowledge and interactions, thus ensuring scalability. For example, in the fire tracking application, each FireTracker agent only needs to check its clockwise and counter-clockwise directions to form a perimeter around the fire. Complex application-wide behaviors like dynamic perimeter formation emerge through these simple local decisions. This phenomena is characteristic of swarm-like systems that exist for other types of networks [11]. Agilla provides a platform on which these distributed algorithms can be implemented.

4.2.6 Adaptation to Node Failures

Node failures are a frequent occurrence in wireless sensor networks. When a node fails, all tuples and mobile agents residing on that node are permanently lost. While this may cause application failure, Agilla provides the capability for applications to self-heal. Specifically, an application can self-heal by cloning or moving its agents onto the replacement node when it is installed. Unlike other in-network reprogramming systems, Agilla gives application developers control over the self-healing process. For example, in the fire tracking application, a node will fail when it catches on fire. The FireTracker agent self-heals by detecting this failure and cloning itself around the failed node to ensure the integrity of the perimeter. Note that in other systems like Trickle [101] and Deluge [77], the developer has less control since the application is always flooded throughout the entire network.

Table 4.1: Memory Availability and Size of Agilla

	Available		Agilla	
Platform	ROM	RAM	ROM	RAM
Mica2	128K	4K	57K	3.3K
TelosB	48K	10K	45K	3.4K

4.2.7 Security

Security is important to some applications of WSNs [141]. Mobile agents can introduce security risks due to their mobility. Existing techniques can be used to achieve various levels of security in Agilla. For example, wireless transmissions can be encrypted [85], and each agent can be enclosed within a "sandbox" [143]. More advanced techniques that are particularly useful in validating untrusted mobile code include code verification [35] and proof-carrying code [129]. The current implementation of Agilla does not integrate all of these mechanisms, but they can be integrated, assuming sufficient resource availability, to achieve the desired level of security.

4.3 Implementation

Agilla's middleware is designed to meet the unique challenges of WSNs like severe resource constraints and unreliable wireless networks. This section first presents the target platform, then describes Agilla's middleware architecture followed by its instruction set architecture.

4.3.1 Target Platform

Agilla was initially implemented on Mica2 nodes [38], which have an 8-bit 7.38MHz Atmel ATmega128L microprocessor and a Chipcon CC1000 radio transceiver. The radio communicates at up to 38Kbps over a range of up to 100m [186]. The nodes have only 128KB of ROM and 4KB of RAM. Since its initial implementation, Agilla has been ported to the MicaZ [39], Tyndall 25mm [170], and TelosB [144] nodes, which have similar characteristics to the Mica2.

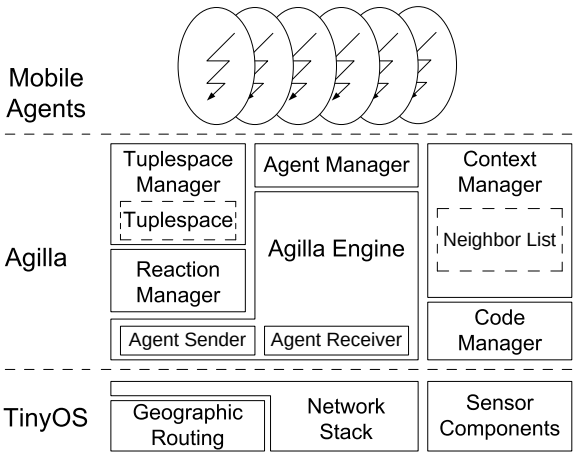


Figure 4.3: Agilla's middleware architecture

The severe resource constraints of WSN nodes require that Agilla be carefully engineered. While the operating system does not affect Agilla's basic programming model, it does affect the implementation. Agilla can be implemented on many different operating systems. For this study, TinyOS [73] is used. TinyOS does not support binary module updates like Contiki [48] or SOS [13]. However, this ability is complementary to Agilla and may enhance Agilla's flexibility. Through the careful engineering described in this section, Agilla was able to fit in the limited resources available, as shown in Table 4.1. Note that, due to limitations imposed by TinyOS, all memory is statically allocated. Thus, Table 4.1 lists the *total* amount of memory consumed by both the middleware and the potential applications executing on top of the middleware.

4.3.2 Middleware Architecture

Agilla's software architecture is divided into three layers, as shown in Figure 4.3. The highest contains the agents and is discussed further in Section 4.3.3. The middle contains the core Agilla middleware components, while the bottom is the operating system.

The middleware consists of several components that are orchestrated by an Agilla Engine, which is the virtual machine (VM) kernel that controls the concurrent execution of all agents on a node. It implements a round-robin scheduling policy. If

Type	Size (Bytes)	Content
State	16	agent id, program counter, code size, condition code, stack pointer
Code	26	one instruction block
Heap	26	four variables and their addresses
Stack	26	four variables
Reaction	26	one reaction

Figure 4.4: Messages used during migration

an agent executes a long-running instruction like `sleep`, `sense`, or `wait`, the engine immediately switches agents.[1] The VM's scheduler is implemented as a TinyOS task that is continuously posted. Each time the task is posted, it executes one instruction. By switching agents in a round-robin fashion, the effect is concurrent execution of all agents. Note that this design is due to TinyOS's execution model, which is based on tasks. If Agilla were implemented on an operating system supporting threads like Contiki or SOS, the VM scheduler could be replaced by multiple concurrent threads each executing a different agent.

The Agilla Engine also handles agent arrival and departure. When an agent migrates, Agilla divides it into multiple types of messages as shown in Figure 4.4. The message sizes are based on restrictions imposed by TinyOS's network stack and do not limit the size of an agent. Multiple messages will be used if necessary. For example, if an agent has 30 bytes of code, two code messages will be used.

The highly unreliable nature of WSN wireless links may prevent agents from reliably migrating. This is a problem since a WSN application may fail or perform poorly if it looses an agent. To address this problem, an agent is migrated one hop at a time, and is acknowledged after each hop. This technique is used by other transport protocols within WSNs to increase reliability [173], but they focus on packet communication in general while Agilla applies the approach to agent migration in particular.

When an agent fails to migrate, it resumes on the sender node with an error flag set. The agent can then perform an application-specific response. While this may result in duplicate agents when the migration was actually successful, the possibility of duplicate agents is usually preferable to losing an agent. Consider the fire detection

[1] A large body of work exists on scheduling policies that provide real-time guarantees or multiple priority levels [162]. These advanced scheduling policies may be incorporated into the Agilla Engine, assuming sufficient resource availability.

application; it is better to have duplicate fire warnings rather than no warnings at all. Of course, having duplicate agents wastes resources. To address this, agents can be programmed to detect when a certain number of identical agents are present and terminate when this threshold is reached. Note that when an agent attempts to migrate a long distance, the agent may be resumed in the middle due to migration failure. The programmer must determine what to do in this situation.

The Agilla Engine is supported by several components including the Agent, Code, Context, Tuple Space, and Reaction Managers. Each manager provides a unique service that collectively forms Agilla's middleware. The Agent Manager maintains agent execution state. The Code Manager dynamically allocates code memory for each agent, and fetches instructions as the agent executes. The Context Manager keeps an updated list of neighboring nodes. The Tuple Space Manager maintains the local tuple space and implements the non-blocking operations (both local and remote). Finally, the Reaction Manager stores the reactions registered by the local agents and determines when a reaction should fire.

Agilla's middleware is highly optimized to reduce memory consumption. Optimized components include the Context, Tuple Space, and Reaction Managers. The Context Manager uses a simple beaconing mechanism for neighbor discovery. Provided enough resources and application requirements, more advanced protocols may be implemented [178, 176, 180]. The Tuple Space Manager does not implement blocking operations, which are implemented within the instructions themselves as described in Section 4.3.4. The Reaction Manager operates asynchronously with the Tuple Space Manager. This simplification reduces memory consumption, but weakens the semantics of reactions slightly by only ensuring that a reaction will *eventually* fire provided the matching tuple remains in the tuple space.

4.3.3 Agent

The architecture of an Agilla mobile agent is shown in Figure 4.5. It consists of a stack, heap, and various registers. A stack architecture is used to enable higher code density. Most Agilla instructions are a single byte. By default, the operand stack is

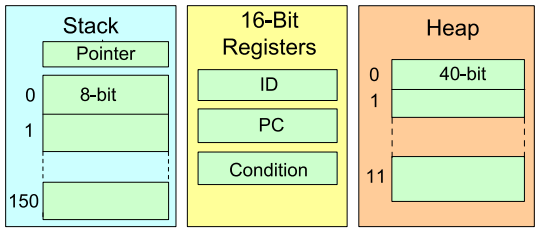


Figure 4.5: The mobile agent architecture

105 bytes, which is small enough to fit on the Mica2 platform. The heap is a random-access storage area that allows each agent to store 12 variables. It is accessed by instructions `getvar` and `setvar`. The operand stack and heap sizes are customizable based on memory availability.

The agent also contains three 16-bit registers: the agent's unique ID, the program counter (PC), and the condition code. The agent ID is unique and is maintained across move operations. A cloned agent is assigned a new ID. An agent ID is generated by concatenating the least significant byte of the host address with a monotonically increasing counter on the host.[2] The PC is the address of the next instruction. Finally, the condition code is a 16-bit register that records execution status. In addition to recording the results of comparison instructions, this register records whether a migration operation is successful, whether the agent is the original or clone, and if it failed, why.

4.3.4 Instruction Set Architecture (ISA)

Agilla's ISA is tailored to the unique properties of self-adaptive WSN applications (e.g., localized interactions), and to the mobile agent computing model (e.g., agent migration and the need for context information). Some of Agilla's unique instructions that achieve this are presented in Figure 4.6. A full listing is available on Agilla's website [52]. Agilla's ISA can be divided into four categories: general purpose, extended, tuple space, and migration. General-purpose instructions enable agents to perform

[2]Our current implementation assumes a maximum of 256 nodes and that each node may clone up to 256 agents. Future implementations that require higher limits may increase the size of the AgentID, or implement a mechanism that recycles old agent IDs belonging to agents that have have already died. To prevent duplicate IDs due to node failure, the counter can be stored in flash memory and restored when the node reboots.

Instruction	Description
`loc`	Pushes the current location onto the stack
`aid`	Pushes the agent's ID onto the stack
`numnbrs`	Pushes the number of neighbors onto the stack
`wait`	Stops agent execution, allows it to wait for a reaction
`sleep`	Sleeps for the value * 8 ms
`smove`	Strong move
`wmove`	Weak move
`sclone`	Strong clone
`wclone`	Weak clone
`getnbr`	Get a specific neighbor's address
`randnbr`	Get a random neighbor's address
`out`	Insert a tuple into the local tuple space
`inp`	Non-blocking find and remove tuple from the local tuple space
`rdp`	Non-blocking find tuple in tuple space
`in`	Blocking find and remove tuple from tuple space
`rd`	Blocking find tuple in the local tuple space
`rout`	Insert a tuple into a remote tuple space
`rinp`	Non-blocking find and remove tuple from a remote tuple space
`rrdp`	Non-blocking find tuple in remote tuple space
`regrxn`	Register a reaction on the local tuple space
`deregrxn`	Deregister a reaction on the local tuple space

Figure 4.6: Noteworthy Agilla instructions

basic tasks like obtaining the neighbor list, sensing, periodically sleeping to conserve energy, and continuously repeating certain application-specific operations. Extended instructions are application-specific. They enable an agent's efficiency to be significantly increased by changing the virtual-native code boundary [99]. For example, in the fire tracking agent, a helpful instruction is one that determines which neighbors are on fire. Defining an extended instruction requires the application developer to implement the instruction and wire it into the VM. Since TinyOS does not support dynamically linked modules, adding an extended instruction can only be done offline. The remainder of this section discusses the tuple space and migration instructions.

Tuple space instructions. Tuple space operations allow an agent to interact with the tuple space on each host. These operations require a tuple or template parameter. This is done by pushing each field followed by the number of fields onto the stack. For example, in Figure 4.2, lines 1-3 push a template with two fields onto the stack.

Instructions `out`, `in`, `rd`, `inp`, and `rdp` access the local tuple space. The blocking `in` and `rd` operations are implemented by having the agent repeatedly try the probing

`inp` or `rdp` operations. If no matching tuple is found, the agent is stored in a wait queue. When a tuple is inserted, the agents in this queue are notified and re-check for a match. The remote tuple space operations `rout` (remote `out`), `rinp` (remote probing `in`), `rrdp` (remote probing `rd`), `routg` (remote group out), and `rrdpg` (remote probing group `rd`) are non-blocking to avoid deadlock due to message loss and disconnection. The group operations are done in a best-effort basis using wireless broadcast. Applications must be engineered with these best-effort semantics in mind. `rrdpg` relies on a time-out to determine when to stop waiting for replies. The tuple space instructions also include `regrxn` (register reaction) and `deregrxn` (de-register reaction).

Migration instructions. Migration instructions allow an agent to move or clone to another node. Agilla provides four migration instructions: `smove` (strong move), `wmove` (weak move), `sclone` (strong clone), and `wclone` (weak clone). Strong operations transfer both the state and code, while weak operations only transfer the code. As mentioned in Section 4.2.1, strong migrations are more powerful since they allow an agent to resume where it left off prior to migrating, but incur higher overhead. Weak migrations are more efficient, but force agents to resume from the beginning upon arrival at the destination.

4.4 Micro-benchmarks

This section presents micro-benchmarks demonstrating feasibility of Agilla's programming model in resource-constrained WSNs.

4.4.1 Micro-Benchmarks

The following micro-benchmarks are performed on a network of 25 Mica2 nodes arranged in a 5x5 grid. To achieve a multi-hop network, messages are filtered based on the grid topology. Geographic routing [88] is used for remote tuple space operations and agent migrations.

```
// The smove agent
1:      pushloc 5 1
2       smove           // move to node (5,1)
3:      pushloc 1 1
4:      smove           // move to node (1,1)
5:      halt

// The rout agent
1:      pushc 1
2       pushc 1         // tuple <value:1> on stack
3:      pushloc 5 1
4:      rout            // do rout on node (5,1)
5:      halt
```

Figure 4.7: The agents that test `smove` (top) and `rout` (bottom)

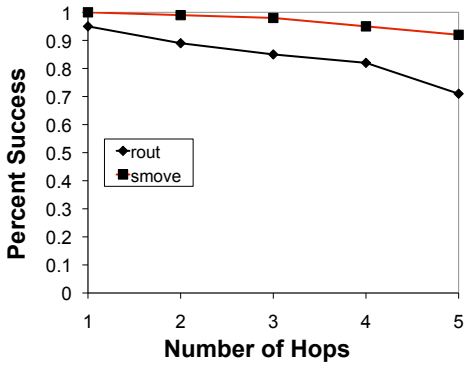


Figure 4.8: `smove` vs. `rout` reliability

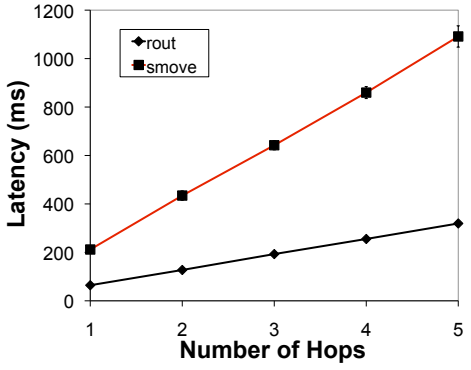


Figure 4.9: `smove` vs. `rout` latency

To test migration and tuple space operations, the agents shown in Figure 4.7 are used. The strong move (`smove`) agent moves to a remote node and back while the remote out (`rout`) agent places a tuple in a remote node's tuple space. The distance between the original and destination nodes is varied from 1 to 5 hops and each experiment is repeated 100 times. The latency of each successful execution and the number of failures are recorded. The results, shown in Figures 4.8 and 4.9, indicate that as the distance increases, the probability of message loss also increases, which is reflected in a decrease in reliability. The reason `rout` is less reliable than `smove` is because, in these experiments, `rout` does not use ARQs for retransmitting lost packets. Note that confidence intervals are not given in Figure 4.8 since it shows the percent success across all experimental rounds. Figure 4.9 contains 95% confidence intervals calculated over the successful rounds (the error margins are difficult to see because they are so small, i.e., $\pm 3.09ms$ for `rout` and $\pm 43.79ms$ for `smove`).

The one-hop latencies of all remote operations are measured by timing each 100 times and finding the average. The results, shown in Figure 4.10, are similar to that for `rout` and `smove` and show that the agent migration instructions have higher overhead than the remote tuple space operations. The figure also shows that the strong and weak migration operations have approximately the same latency. This is because the additional overhead of transmitting a few more packets containing the agent's state is small relative to the cost of performing hop-by-hop migration. Note that the migration latencies have higher variance due to the use of ARQ. The results suggest that an agent can reliably migrate one hop every 0.3s on the Mica2 platform. Assuming the radio range is 50m, this means an agent can migrate across a network at 600km/h (373mph), which is sufficient for tracking many interesting phenomena like fire.

Local operations unique to Agilla are also benchmarked. Each local instruction is benchmarked 100 times, and the average execution time is calculated. The results, shown in Figure 11, indicate that local operations execute quickly relative to remote operations. We did not directly compare Agilla's instructions with other WSN VMs like Maté [98] because many of Agilla's instructions are higher level and do not have a corresponding instruction against which to compare. However, the latency of simpler Agilla instructions that execute within 100μs like get location (`loc`) and get agent ID (`aid`) are comparable to corresponding Maté operations [98].

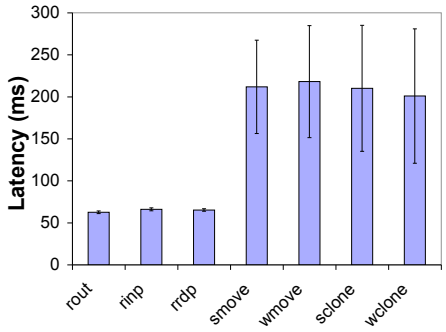


Figure 4.10: Remote operation lantency

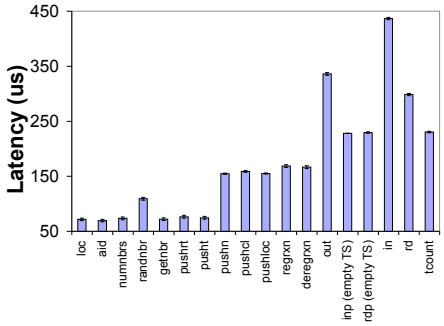


Figure 4.11: Local operation latency

In summary, Agilla can perform one-hop remote tuple space operations within 55ms, and migration operations in 225ms per hop on Mica2 nodes. The migration latency scales linearly with the number of hops, and the additional overhead for reliable operations is justified by their resilience to message loss across multiple hops. This demonstrates the feasibility and efficiency of using mobile agents and tuple spaces in a WSN.

The remainder of this section compares Agilla with two existing systems that provide in-network reprogramming, Maté [98] and Deluge [77]. Maté provides a virtual machine that enables interpreted code updates, while Deluge enables native code updates. Comparing Agilla to Maté and Deluge is not straightforward because of their fundamentally different programming models. In Maté and Deluge, underlying services handle the epidemic dissemination of code throughout the WSN, resulting in *every* node running the *exact same* code. Application developers have no control over where their code is installed. Agilla takes the opposite approach in which developers have *total control* over where their code is installed, since mobile agents *explicitly control* their migration patterns. Comparing Agilla to Maté and Deluge requires deploying an application that can be implemented in all three systems. In this case, the application pseudocode is shown in Figure 4.12. It periodically takes the temperature and notifies the base station if it is above a threshold. Since Maté and Deluge automatically spread the program throughout the WSN, the Agilla implementation must do the same, but in the application layer using Agilla's higher-level primitives. Note that network-wide reprogramming is *not* in the spirit of Agilla's programming

```
1:  repeat every second
2:    if (temperature > 100C) then
3:      toggle red LED
4:      send warning to base station
5:    else
6:      toggle green LED
7:    end if
8:  end repeat
```

Figure 4.12: Test program pseudocode

model, and is done only to compare the Agilla middleware's overhead against that of Maté and Deluge. Thus, this is not an evaluation of Agilla's programming model.

The simplest implementation of epidemic code propagation using Agilla is to have an agent immediately clone itself onto each neighbor upon arrival at a particular node. This is inefficient since multiple copies of the agent may end up on the same node. To avoid this, when an agent arrives, it first inserts a tuple containing one field, "mrk", into the local tuple space. This tuple indicates that the local node contains the agent. Then, for each neighbor, it first checks for the presence of this tuple using a remote probing read (`rrdp`). If the neighbor already has a "mrk" tuple, the agent does not migrate to it. This reduces the likelihood that a node will end up with multiple copies of the agent.

The code size of each implementation is as follows: Maté 26 bytes, Agilla 41 bytes, and Deluge 24,866 bytes. The sizes of the Maté and Agilla implementations are platform independent since both are based on virtual machines. The Deluge implementation is platform-dependent and shows the size on the TelosB platform [144]. The code sizes indicate that Agilla and Maté are comparable with Agilla taking slightly more memory due to the application-level implementation of the epidemic spreading protocol. The Deluge implementation is significantly larger since it includes the entire TinyOS, which must be re-transmitted each time a new program is deployed. This is a function of TinyOS, which does not support the dynamic linking of modules. An implementation on a different operating system like SOS that does support dynamic linked modules will not require transferring the entire OS, but will most likely result in larger code size than Maté or Agilla due to the use of lower-level instructions.

To evaluate the performance of the in-network reprogramming systems, a testbed consisting of 31 TelosB nodes deployed throughout the fifth floor of the computer

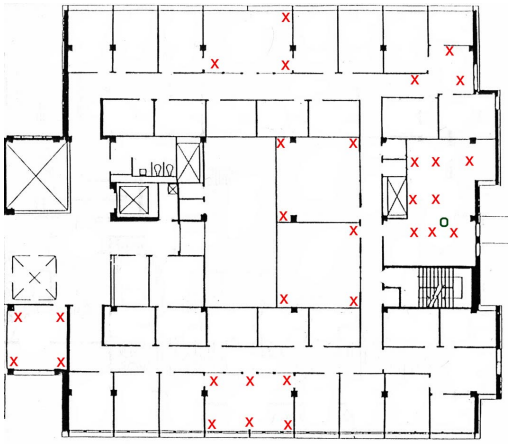


Figure 4.13: The layout of the WSN testbed consisting of 31 TelosB nodes spread across 1000 square meters. Red 'x' indicate node placement, the green 'o' marks the gateway.

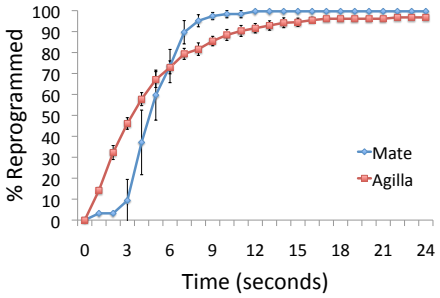


Figure 4.14: Maté and Agilla reprogramming rates

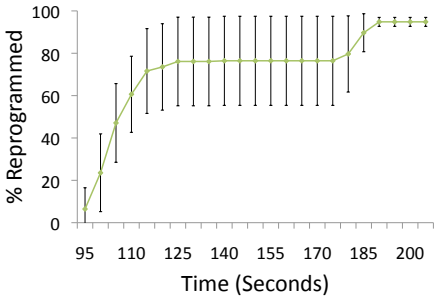


Figure 4.15: Deluge reprogramming rate

science building at Washington University in St. Louis is used. The testbed's layout is shown in Figure 4.13. The TelosB nodes are represented by red 'x' marks and the maximum network distance between any two nodes is three hops. The gateway node is indicated by the green 'o', and is where the application is initially injected. The testbed is instrumented with 8 Linksys NSLU2 micro-servers that have direct USB connectivity to every TelosB node in the network and wired Ethernet connectivity to a central server. This infrastructure is used to record when the application arrives on each node. For each in-network reprogramming system, the program shown in Figure 4.12 is injected ten times, and the rate of propagation is recorded. The results of the experiments are shown in Figures 4.14 and 4.15.

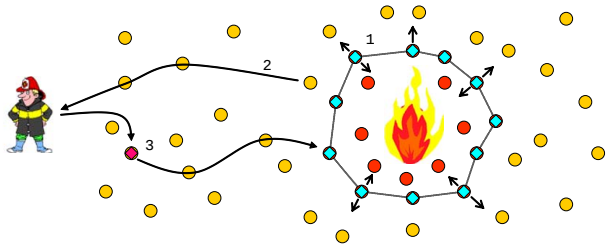


Figure 4.16: An overview of the fire detection and tracking application. When a fire breaks out, detection agents sense the fire (1) and send a message to a base station (2), which injects a tracker agent into the network (3). This agent migrates to the fire and clones itself to form a perimeter. The perimeter is continuously adjusted based on the fire's behavior.

Figure 4.14 shows that both Agilla and Maté are able to program the network quickly, with Maté achieving 98 ± 2% coverage within 10s and Agilla achieving 94 ± 6% coverage within 14 seconds. This makes sense since Agilla implements a relatively simple spreading algorithm at the application level, whereas Maté natively implements the sophisticated Trickle [101] algorithm. Figure 4.15 shows that Deluge takes significantly longer to reprogram the testbed since it has to transfer the entire program and operating system. It is able to program 90 ± 10% of the network in 185 seconds. The results show that Agilla and Maté perform comparably, and both are significantly faster than Deluge.

4.5 Application Case Study: Fire Detection

This section evaluates how Agilla enables developers to create highly adaptive applications in a dynamic environment. A fire detection and tracking application is used, as shown in Figure 4.16. In this application, a WSN is deployed in a region susceptible to fires. `FireDetection` agents are deployed and patrol the network for fires. When they detect a fire, they notify a `FireTracker` agent, which moves to the fire and repeatedly clones itself to form a perimeter. The tracker agents autonomously adjust their numbers and locations to maintain a perimeter as the fire changes shape. The remainder of this section presents the implementation and evaluation of the fire detection and tracking application. The application consists of three types of agents: `Fire` agents that emulate fire, `FireDetection` agents, and `FireTracker` agents.

```
1:  BEGIN       pushn fir
2:              pushc 1
3:              out                 // insert fire tuple
4:  BLINK_RED   pushc 25
5:              putled              // toggle red LED
6:              pushc 1
7:              sleep               // sleep for 1/8 second
8:              rjump BLINK_RED
```

Figure 4.17: The static `Fire` agent

```
1:  BEGIN       pushn fir
2:              pushc 1
3:              rdp                 // Search for fire tuple
4:              rjumpc FIRE         // jump if fire tuple found
5:  SLEEP       pushc 8
6:              sleep               // sleep for 1 second
7:              rjump BEGIN         // repeat
8:  FIRE        pop
9:              pop                 // pop off fire tuple
10:             loc
11:             pushn fir
12:             pushc 2             // tuple <"fir", location>
13:             pushloc 1 1         // assume fire tracker is at (1,1)
14:             rout                // send tuple to fire tracker
15:             halt                // die
```

Figure 4.18: A `FireDetector` agent

Fire Agents

`Fire` agents emulate fire by inserting the string "fir" into the local tuple space. Fire is detected by searching for these tuples. Two types of `Fire` agents are used: *static* and *dynamic*. A static `Fire` agent does not move and serves as a baseline on how quickly the `FireTracker` agent can form a perimeter around a fire. Its code is shown in Figure 4.17. Lines 1-3 insert the fire tuple, while lines 4-8 continuously blink the red LED. By monitoring the LEDs, the application's state is determined. Dynamic `Fire` agents model a fire that spreads. It is implemented in a mere 47 bytes of instructions and is available on Agilla's website [52].

```
1:  REG_RXN      pushn fir
2:               pushc 1
3:               pushc  RXN_FIRED
4:               regrxn              // register the reaction
5:               ...                 // tracking code omitted
6:  RXN_FIRED    pushc 9
7:               putled              // turn off LEDs
8:               pushn trk
9:               pushc 1
10:              inp                 // remove tracker tuple
11:              halt                // die
```

Figure 4.19: The reaction registered by the `FireTracker` agent

Fire Detection Agents

`FireDetector` agents search for fire tuples, and notify the `FireTracker` agent when a fire is found.[3] The agent's code is shown in Figure 4.18. It searches for a fire tuple once per second (lines 1-7), and inserts a tuple containing its location and the string "fir" onto the node hosting the `FireTracker` agent if a fire tuple is found (lines 8-14). After it notifies the base station, it dies, freeing its resources (line 15).

Tracking Agents

`FireTracker` agents form and maintain a perimeter around the fire. They insert a tuple containing the string "trk" into the local tuple space. Each `FireTracker` agent periodically checks its clockwise and counter-clockwise neighbors (relative to the position of the fire) for this tuple. Note that this is possible because Agilla enables nodes to be addressed by location, meaning the relative position of a neighbor is known. If they exist, the perimeter is considered intact. Otherwise, the agent clones itself onto the neighbor that is missing the tuple to reestablish the perimeter. This is repeated until the perimeter is fully formed. If the perimeter is breached, the `FireTracker` agents next to the breach will detect the missing neighbor and attempt to re-form the perimeter via cloning. Note that perimeter formation is an emergent behavior based on local decisions, which ensures scalability. The actual

[3]In a real deployment, fire can be detected using sensors that give more details about the fire. This would enable more sophisticated fire-tracking algorithms that consider the unique characteristics of the fire.

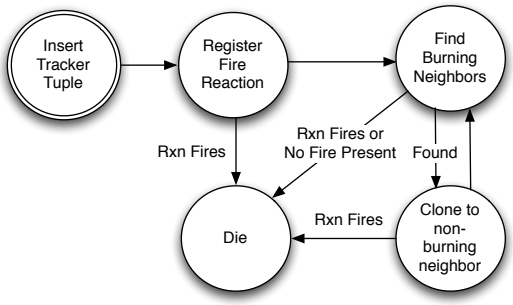


Figure 4.20: The life cycle of a `FireTracker` agent

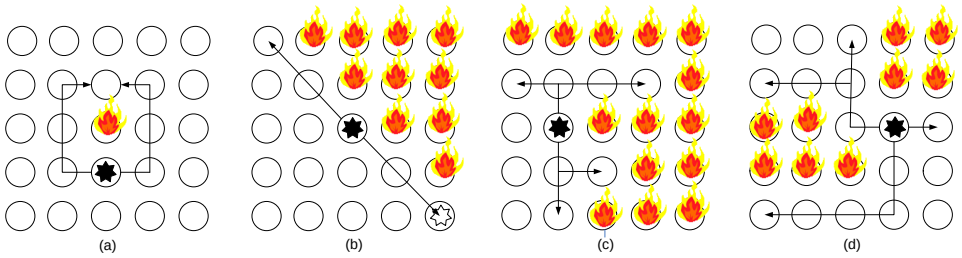


Figure 4.21: The static fire test scenario, the star is the initial position of the `FireTracker` agent

code for creating and maintaining this perimeter is omitted due to space constraints and begins on line 5 of Figure 4.19. It consists of 58 lines of code, and is included with Agilla's distribution [52].

If a node hosting a `FireTracker` agent catches on fire, the `FireTracker` agent must die. This is achieved by registering a reaction that kills the agent when a fire tuple appears. The code for this is shown in Figure 4.19. Lines 1-4 register a reaction sensitive to fire tuples, while lines 6-11 define the call-back function, which turns off the LEDs (lines 6-7), removes the tracker tuple (lines 8-10), and kills the agent (line 11).

The life cycle of a `FireTracker` agent is shown in Figure 4.20. It works by repeatedly checking whether the perimeter is breached, and cloning itself to re-create the perimeter if necessary. The periodic checking allows the perimeter to be adjusted as the fire spreads, thus achieving self-adaptation. The `FireTracker` agent was implemented in only 101 bytes of code, demonstrating the expressiveness of the Agilla programming model.

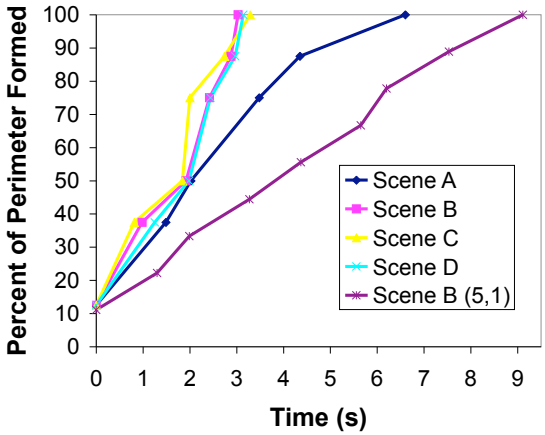


Figure 4.22: The rate of forming a perimeter around static fires

The `FireTracker` agent was evaluated using both static and dynamic fires, using the same network as was used in the micro-benchmarks. For the static fire tests, static `Fire` agents are used to form fires of various shapes, and a `FireTracker` agent is injected onto a node next to the fire. The time to form a perimeter is recorded. The types of fires used are shown in Figure 4.21. The starting location of the `FireTracker` agent is marked with a black star, and the arrows indicate where the agent must clone to form the perimeter. Note that in scene B, node (5,1) in the lower-right corner also has a star. This is to evaluate how the starting location of the `FireTracker` agent impacts efficiency.

The results of the static fire tests are shown in Figure 4.22. In most cases the perimeter is formed within 3 seconds. Scene A took longer because its shape limits the number of `FireTracker` agents that can spread in parallel. For example, when a `FireTracker` agent is at node (2,2), which is in the lower-left corner, it is the only agent that can clone to (2,3). To test this, we re-ran scene B with the `FireTracker` agent initialized at node (5,1), which is in the lower-right corner. The results, shown in Figure 4.22, indicate that the initial point of `FireTracker` significantly impacts performance.

To evaluate Agilla's ability to maintain a perimeter around a spreading fire, four `FireDetector` agents are injected into the network at the positions marked with a star in Figure 4.23. A dynamic `Fire` agent is then injected into node (5,5). Note that `FireDetector` agents must first detect the fire and notify the `FireTracker`

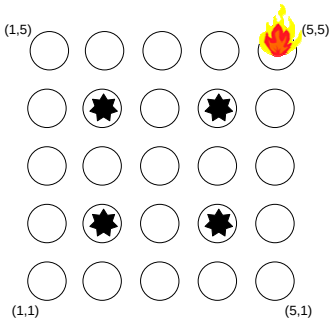


Figure 4.23: Dynamic Fire Test Settings

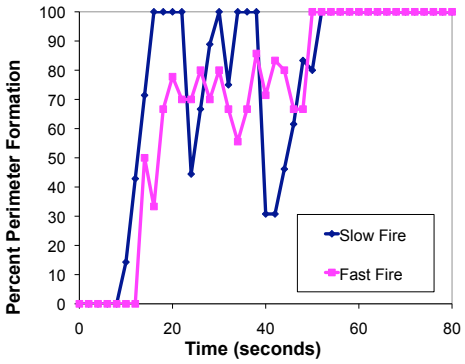


Figure 4.24: Dynamic Fire Perimeter Formation

agent before perimeter formation can begin. The `FireTracker` agent is initially at (1,1). Two tests are run: one with a slow `Fire` agent that spreads one-hop every 7 seconds, another with a fast one that spreads every 5 seconds. The results, shown in Figure 4.24, indicate that the `FireTracker` agent does a reasonable job maintaining a perimeter around the slow fire, but has difficulty with the fast one due to the network being partitioned by the fire. The reason why both converge to 100% is because as the fire spreads, the network eventually becomes saturated. While it is true that in our limited network, every node will eventually be on fire, the `FireTracker` agents were able to at least partially form a perimeter long before every node is engulfed.

This case study shows that Agilla provides a convenient programming model for implementing highly adaptive applications in a dynamic environment. With Agilla, we created a non-trivial self-adaptive application (fire-tracking) with minimal effort, while achieving sufficient application-level performance.

4.6 Application case study: Robot Navigation

Consider a robot that needs to travel through a region while avoiding fires, as shown in Figure 4.25. Since the robot's on-board sensors have limited range, it uses the WSN and a navigation service [22] to find a safe route around the fire. To ensure efficiency, the possible routes are restricted to those on a roadmap [19], which is shown as an

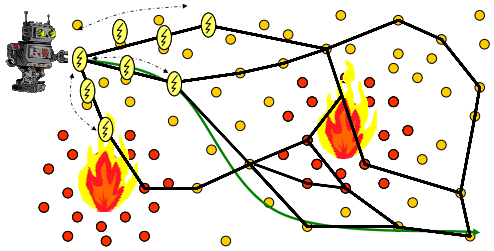


Figure 4.25: The robot navigation problem. A roadmap graph is overlaid on the WSN and mobile agents are used to query the temperature along the edges, which helps the robot navigate around the fires.

overlay graph in Figure 4.25. The problem is how to determine which paths in the roadmap are safe to traverse.

Assuming safe paths have maximum temperatures below a certain threshold, the robot deploys network exploration agents to determine the maximum temperature along a path. Once injected, the agent repeatedly clones itself onto nodes along the path and sends back temperature information. To limit overhead, the agents spread themselves out and filter data coming from further down the edge by filtering all temperature readings less than the local temperature. Once the robot receives the maximum temperature along the edges of the overlay graph, it can decide which route to take.

The navigation service was implemented in only three days by developers who were initially not familiar with Agilla's middleware and evaluated using a testbed consisting of 17 Mica2 motes and an ActiveMedia Pioneer-3 DX robot [22]. Empirical results demonstrate the feasibility of using Agilla in this highly dynamic application. The implementation demonstrates how Agilla enables non-trivial in-network processing techniques to be implemented as mobile agents.

4.7 Agimone: Integrating Wireless Sensor Networks with IP Networks

The ability to create applications consisting of mobile agents that can migrate among WSN devices and explicitly control their movements brings about the possibility of expanding the domain in which the agent can migrate. For example, suppose an agent can originate in WSN `A`, perform some computations, and then migrate onto WSN `B` to complete the necessary computations. This is useful because many WSN applications involve sensing regions too great or disjoint to be covered by a single WSN.

For example, consider cargo container monitoring. In this application, WSN nodes are deployed on cargo containers to monitor them as they travel around the world. The motivation may be security, i.e., to ensure that the container is not opened enroute, or contractual, i.e., to ensure the container is not exposed to temperatures above a certain threshold. Cargo containers are continuously moved by many different forms of transportation, including air, ship, rail, and truck. When they are not being moved, they often reside at warehouses or shipping yards. Each of these modalities involve physically separate WSNs. Having a system that enables a user to treat the numerous disjoint WSNs as a single logical network is powerful, as it could orchestrate WSNs scattered around the world to cooperate to acheive a common task. For example, it would allow a user to search for a particular item among all cargo containers belonging to a particular company, despite the fact that the containers are physically scattered around the world.

Integrating WSNs with IP networks like the Internet can be done using gateway devices like the Stargate [161], which translate between the WSN and IP network. The key challenge is to develop a software framework that seemlessly integrates the two forms of networks into one logical network. Two middleware frameworks that appear to be good candidates for acheiving this include Agilla and Limone. Agilla provides mobile agent and tuple space abstractions in WSNs, while Limone provides similar abstractions in MANETs, which are often based on IP.

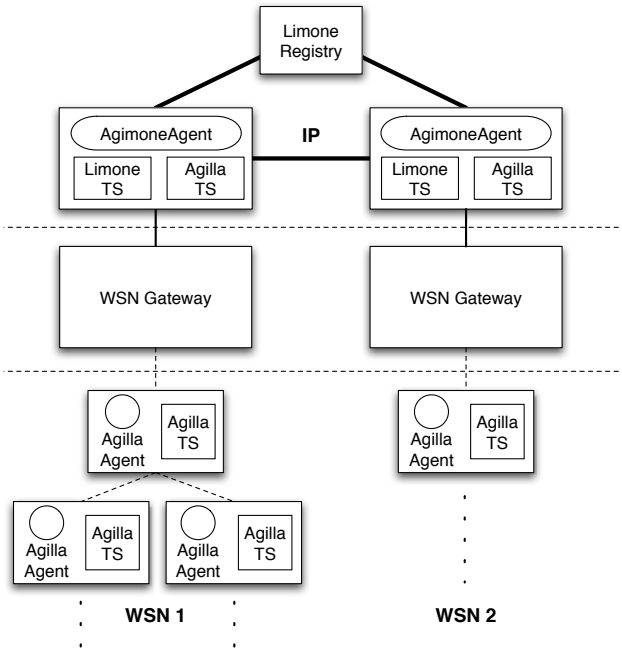


Figure 4.26: The Agimone system architecture. Each WSN has a gateway running Limone. The gateways communicate using Limone, forming a bridge for Agilla agents to migrate between WSNs. A Limone registry is used to discover available WSNs

However, integrating the two middleware frameworks is not straightforward because of the differences in the underlying platforms for which they were designed, resulting in varying APIs and capabilities. For example, Limone agents maintain their own tuple spaces while Agilla agents do not. In addition, Limone addresses a tuple space based on the ID of the agent that owns it, whereas Agilla addresses tuple spaces based on the location of the device that holds it. These differences require a software translation layer between the two middleware systems. Traditionally, bridging gaps between two middleware systems is application-specific and a costly and error-prone process. For this reason, ***Agimone*** is developed that joins Agilla and Limone via a thin but reusable integration layer, enabling applications to span both IP and sensor networks.

4.7.1 System Architecture

Agimone's system architecture is shown in Figure 4.26. It consists of multiple WSNs running Agilla, each with a gateway running Limone. The gateway has a Limone agent called `AgimoneAgent`, which acts as the arbiter between the WSN and IP networks. It is responsible for (1) discovering the presence of remote WSNs, (2) advertising these WSNs to agents within the WSN network, (3) forwarding incoming agents from the WSN to the gateway of the destination WSN network, (4) injecting agents arriving from remote WSNs into the local WSN, and (5) serving as a translator between Limone and Agilla tuples.

The process of discovering remote WSNs can be done using a number of different methods. One is to use simple beacons, assuming multicast routing between the WSN gateways is enabled. Another is to use a centralized service registry, which is what the current prototype system uses, as shown in Figure 4.26. The actual system used for remote WSN discovery is interchangeable, so more sophisticated protocols like Bonjour [32] can be used if necessary.

The version of Limone running on the gateway has a special tuple space called `AgillaTS`. This tuple space adheres to the tuple formatting and API specifications of the tuple spaces used by Agilla. It enables communication between IP networks and WSNs. Through this tuple space, Limone agents can access sensor data produced by the WSN, and Agilla agents can access computational resources available on the IP network. To seemlessly integrate `AgillaTS` into Limone's programming model, it is revealed to `AgimoneAgent` as just another tuple space belonging to a normal agent residing on the local device. The agent that owns the `AgillaTS` is actually hidden because, in reality, it is virtual and only appears in `AgimoneAgent`'s acquaintance list.

For a Limone agent to send a tuple into the WSN, it simply inserts the tuple into `AgimoneAgent`'s tuple space. `AgimoneAgent` will take this tuple, translate it into the format used by Agilla, and insert it into `AgillaTS`. Likewise, whenever an Agilla agent wishes to send a tuple to a Limone agent, it simply places the tuple into the `AgillaTS` using a remote invocation to special location (UART_X, UART_Y). `AgimoneAgent` will take any tuple placed in `AgillaTS`, translate it into Limone's tuple format, and place it in its own local tuple space. Thus, `AgimoneAgent` ensures that its own tuple

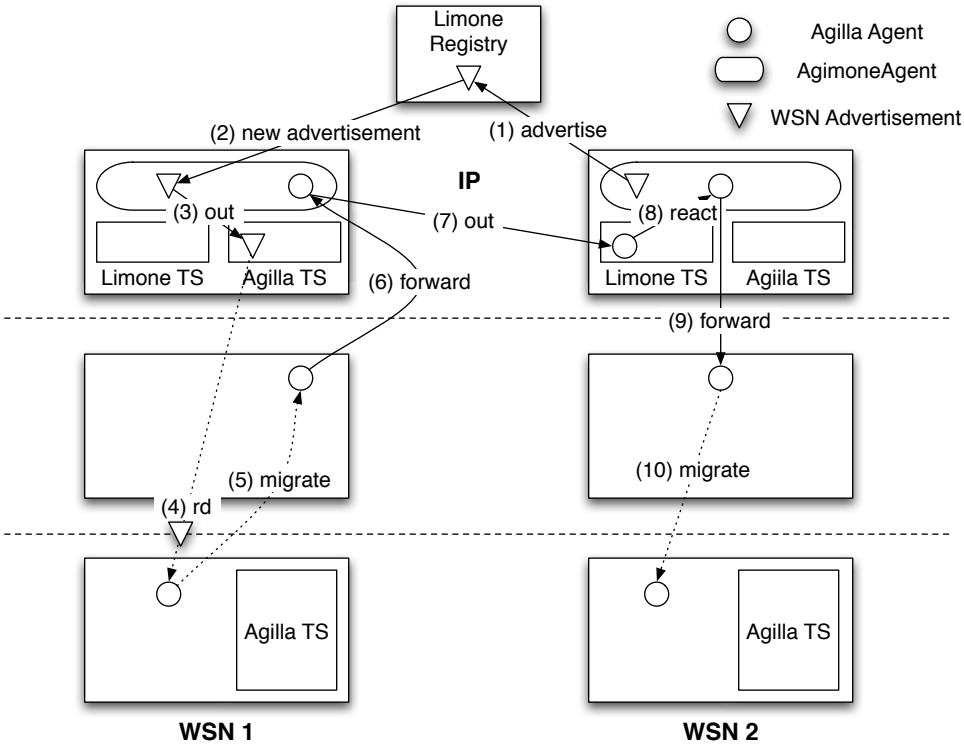


Figure 4.27: Agilla Agent Migration Across Different WSNs

space and `AgillaTS` are mirror-images of each other, enabling communication across middleware boundaries.

The process of translating between Agilla and Limone tuples is non-trivial because Agilla imposes many restrictions on the types of data a tuple can contain. For example, if a Limone agent tries to place the tuple <"mark", 1> into the `AgimoneAgent`'s tuple space, an error condition will arise since Agilla does not support tuple fields containing strings longer than three characters due to memory constraints. To resolve this problem, `AgimoneAgent` uses Limone's tuple space operation manager described in Section 3.2 to filter incoming tuple space operations to those involving tuples that are compatible with Agilla. The developer of the Limone agent must be aware of Agilla's limitations for communication to occur between Limone and Agilla agents.

4.7.2 Migration Across WSNs

In addition to enabling communication between Limone and Agilla networks, Agimone also facilitates Agilla agent migration between two disjoint WSNs. This enables applications to seamlessly execute across multiple WSNs. Before an agent can migrate into a different WSN, it must first discover the identity of the destination WSN. To achieve this, `AgimoneAgent` publishes a tuple describing properties of the WSN, like its location, on the Limone Registry, which distributes it to all other `AgimoneAgents` in the system. Upon receiving this tuple, `AgimoneAgent` places the tuple into the local `AgillaTS`, thus revealing it to the Agilla agents within the WSN. This is shown as steps 1-4 in Figure 4.27. Referring to the remaining steps in Figure 4.27, when an Agilla agent wishes to migrate, it performs a remote **rd** operation on `AgimoneAgent`'s tuple space to obtain the advertisement tuple (step 4). Using this tuple, the Agilla agent migrates onto the gateway (step 5), where it is encapsulated in a Limone tuple and forwarded to `AgimoneAgent` (step 6). Upon receiving this tuple, `AgimoneAgent` sends it to the destination WSN (step 7). The `AgimoneAgent` at the destination reacts to the tuple (step 8), extracts the Agilla agent, and injects it into the destination WSN (steps 9 and 10). This concludes the process of an agent migrating between two WSNs.

4.7.3 Evaluation

Agimone is evaluated by deploying it on two WSNs connected by an IP network. The WSNs are composed of Mica2 motes and are separated by using different radio channels. Each WSN has a single gateway attached to an IBM R40 laptop via a 115.2Kbps serial link. The laptops are connected via a 100Mbps wired Ethernet link. Since they are on the same subnet, discovery is performed using multicast beacons rather than a Limone Registry. The laptops are configured with a 1.5GHz Intel Pentium M processor, 512MB of RAM, Windows XP and Java Standard Edition 5.0. Latencies are measured using Java's `System.nanoTime()` method, which uses the system's most accurate timer. This section presents micro-benchmarks examining the primitives that cross network boundaries. The benchmarks can be divided into three categories: tuple space operations, agent migration, and overall performance.

Operation (Mote-to-PC)	latency (ms)
`rinp`	10.64 ± 0.15
`rrdp`	10.35 ± 0.06
`rout`	10.37 ± 0.07

Operation (PC-to-Mote)	latency (ms)
`rinp`	10.98 ± 0.17
`rrdp`	11.26 ± 0.19
`rout`	10.85 ± 0.07

Figure 4.28: The Latency of Remote Tuple Space Operations

Tuple Space Operations

In these benchmarks, the cost of tuple space operations that cross middleware boundaries is measured. Specifically, the following operations are evaluated: **rinp** (remote probing remove tuple), **rrdp** (remote probing read tuple), and **rout** (remote insert tuple). These operations can be executed in both directions; they may be performed by the `AgimoneAgent` on the tuple space belonging to the WSN gateway (PC-to-Mote), or by an Agilla agent on the base station's tuple space (Mote-to-PC).

Mote-To-PC. The first set of benchmarks determine the latency of an Agilla agent on the WSN gateway accessing `AgimoneAgent`'s Agilla tuple space. Three benchmark agents are used, each of which performs one of the three remote tuple space operations being evaluated (**rinp**, **rrdp**, and **rout**) 100 times over which the average was calculated. The average of these benchmarks, shown in Figure 4.28, indicate that the operations have an average latency of 10 to 11 ms.

PC-To-Mote. The second set of benchmarks repeats the same operation in the opposite direction: the `AgimoneAgent` performs operations on the WSN gateway's tuple space. In this case, since the latency can be directly measured, each experiment calculates the latency of one operation execution. Figure 4.28 shows the average results from 100 runs of each benchmark. Like in the Mote-to-PC benchmarks, the average latency of PC-to-Mote tuple space operations is 10 to 11 ms.

Agent Migration Operations

As discussed in Section 4.7, agent migrations enable agents located in one WSN to migrate across an IP network into another WSN. From an Agilla agent's perspective, an inter-WSN agent migration occurs in an atomic step, e.g., by executing the **smove** operation. However, as discussed in Section 4.7.1, there are in reality many steps

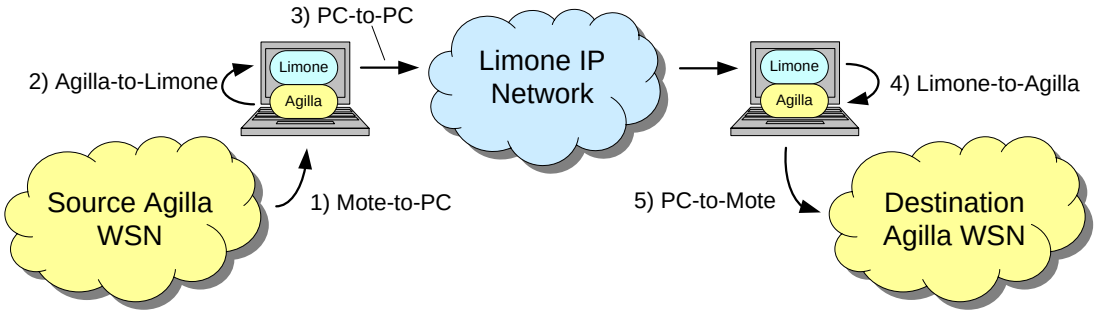


Figure 4.29: The Five Stages of an Inter-WSN Agent Migration Operation.

Stage	Name	Unfiltered Latency	Filtered Latency	# Points Filtered
1	Mote-to-PC	36.12 ± 1.19ms	35.56 ± 0.58ms	2
2	Agilla-to-Limone	1.03 ± 0.16ms	$307.11 \pm 1.59\mu$s	182
3	PC-to-PC	19.45 ± 0.26ms	19.11 ± 0.15ms	19
4	Limone-to-Agilla	1.13 ± 0.16ms	$830.05 \pm 2.26\mu s$	12
5	PC-to-Mote	28.16 ± 5.92ms	23.72 ± 0.56ms	3

Figure 4.30: The Latency of Each Agent Migration Stage (Average of 1000 Runs)

involved which are transparent to the agent. To provide greater insight into the cost of agent migrations, the migration steps shown in Figure 4.27 are grouped into the five distinct stages shown in Figure 4.29. Each state is benchmarked separately. A brief description of these stages and the results of these benchmarks are discussed below.

The results of these benchmarks are collected in Figure 4.30. All benchmark results are presented as an average of 1000 runs. Note the column listing the number of points filtered. These were outlier points with values orders of magnitude above the mean. They are most likely caused by inaccuracies in Java's `System.nanoTime()` method and garbage collection. Since these points are relatively sparse, we filtered them out when presenting the details of each stage below.

Stage 1: Mote-to-PC. In this stage, the agent moves from the source mote to the base station. This procedure is measured by deploying an agent that searches the `AgimoneAgent`'s tuple space for a WSN advertisement and then attempts to migrate to the base station. The average latency of this stage is 35.56 ± 0.58ms.

Stage 2: Agilla-to-Limone. In the second stage, the agent is passed from the Agilla middleware on the base station to the Limone middleware. The cost of crossing middleware boundaries should be negligible, since it only involves a few local method calls. This is borne out by our empirical measurements; the average latency is $307.11 \pm 1.59\mu$s.

Stage 3: PC-to-PC. In this stage, the `AgimoneAgent` encapsulates the migrating agent into a Limone tuple and places it in the destination `AgimoneAgent`'s tuple space. This stage is timed by repeatedly migrating an agent between two base stations, then halving the round-trip time. This stage had an average latency is 19.11 ± 0.15ms.

Stage 4: Limone-to-Agilla. In this stage, the `AgimoneAgent` extracts the encapsulated agent from the Limone tuple and passes it to Agilla's `AgentInjector`. As in stage 2, this process only involves a few local method calls, so the latency should be negligible. The time between placing the tuple in the tuple space to passing the agent to the `AgentInjector` is recorded. The average latency is $830.05 \pm 2.26\mu$s; as expected, this is negligible relative to the other stages.

Stage 5: PC-to-Mote. In the final stage, the agent is injected into the destination WSN. Similarly to stage 1, we measured this latency by migrating an agent which immediately reads an advertisement tuple from the base station, and measuring the time between injection and receiving the tuple space request. The average latency of this stage is 23.72 ± 0.56ms.

Overall Performance

The following benchmarks evaluate the latency of common sequences of operations. The In-and-Out benchmark measures the minimum amount of time it will take for an agent to enter a WSN and return. In a real-world scenario, the agent will perform some application-specific operations while in the network. However, for evaluation purposes, we inject an agent that immediately migrates back onto the base station after it arrives on a mote. The second set of benchmarks (End-to-End) evaluates how long it takes for an agent residing in one WSN to migrate into another WSN and back. The previous section analyzed each stage of the migration process individually; the In-and-Out and End-to-End delays should be the sum of their individual stages. These

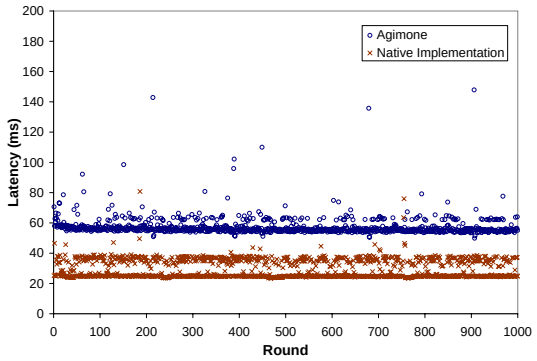


Figure 4.31: The In-and-Out Agent Migration Latency.

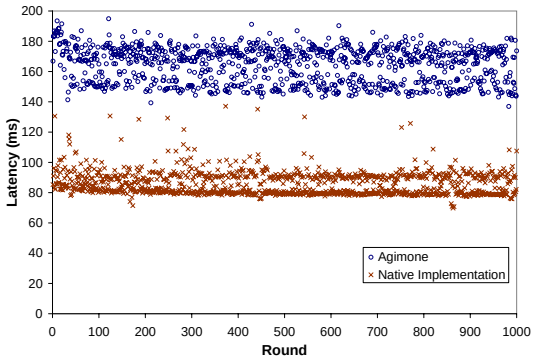


Figure 4.32: The End-to-End Migration Latency

benchmarks validate the results of previous benchmarks. Again, unless otherwise stated, the results presented are the average of 1000 runs.

While Agimone simplifies programming and increases network flexibility, its use of virtual machines results in some overhead. To quantify this overhead, a native-code implementation of In-And-Out and End-to-End is tested and the results are plotted along with the Agimone results. To isolate the cost of message-passing from execution, the native implementations exchange 36-byte data messages wherever the 36-byte mobile agents would migrate.

All of the previous benchmarks used the same 36-byte agent. To gain insight into how the size of the agent affects inter-WSN migration latency, the Exploding-Agent benchmark repeats the End-to-End experiment with successively larger agents.

In-and-Out. This benchmark injects an agent that migrates repeatedly between two WSNs, and measures the cost of moving the agent in and out of one WSN. When the agent is injected into the WSN, it immediately performs a **rrdp** to find the other WSN's advertisement, and then attempts to migrate to it. Thus, this benchmark measures the aggregate of the Mote-to-PC, PC-to-Mote, Limone-to-Agilla, and Agilla-to-Limone migration operations, and the Mote-to-PC tuple space operation. The results of this benchmark are shown in Figure 4.31. The average In-and-Out latency is 62.18 ± 6.09ms, or 56.56 ± 0.27ms when outlying points are filtered out. This is approximately the aggregate of the constituent stages (stages 1, 2, 3, and 4).

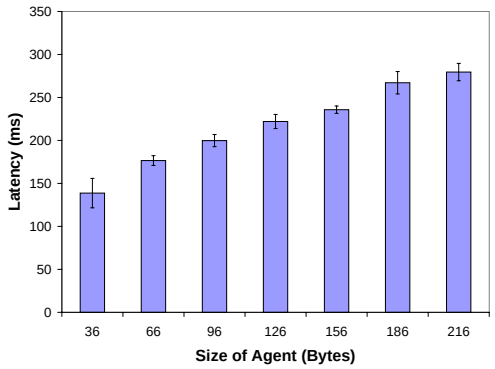


Figure 4.33: End-to-End Latency vs. Size of Agent

The native implementation of In-and-Out is a Java application that queries the base station's attached gateway sensor by sending it two TinyOS packets totalling 36 bytes; the sensor immediately sends 36 bytes of data back. The benchmark measures the time from sending the request to receiving the response. The native implementation has an average latency of 30.09 ± 0.51ms, which is 27.39ms faster than the Agimone implementation.

End-to-End. The End-to-End latency is measured by injecting the agent described above and recording its round-trip time over the IP network. The results are shown in Figure 4.32. The average end-to-end round trip time is 179.19 ± 9.96ms unfiltered, and 164.75 ± 0.96ms filtered. This closely matches the aggregate of the various stages involved.

The native implementation of End-to-End adds to the In-And-Out benchmark by sending a 36-byte packet over the IP network to a remote base station after receiving a response from the WSN. The remote base station sends a 36-byte reply. The benchmark measures the time from querying the sensor node to receiving a response from the remote base station. The native implementation has an average latency of 86.36 ± 2.15ms, which is 78.39ms faster than the Agimone implementation.

Exploding-Agent. This benchmark repeats End-to-End with successively larger agents. The agent's round-trip time is recorded as it bounces between two WSNs 10 times. The results are shown in Figure 4.33. The cost of agent migration is roughly linear with the agent's size and is increasingly dominated by the slow mote-to-mote and mote-to-PC stages.

The benchmarks presented in this section provide a general overview of Agimone's performance and overhead. All inter-network tuple space operations, regardless of direction, take about 10.5ms. A mobile agent takes about 82.5ms to migrate from one WSN to another. Of this, approximately 60ms is spent moving to and from the WSN and its base station, and 20ms is spent traversing the IP network. The latency of migrating into a WSN and back is about 60ms. Of this, most of the time (>57ms) is spent on the serial link between the base station and WSN gateway. The actual transition from Agilla to Limone takes about 307μs, while going from Limone to Agilla takes about 830μs. The overhead of Agimone compared to native code varies depending on the task. In the two operations presented, In-And-Out and End-to-End, there was a 27.39ms and 78.39ms increase in execution time relative to native code, respectively. Native code, however, is not nearly as flexible as mobile agents, and presumably requires more development time. Finally, an agent's migration latency increases linearly with its size.

4.7.4 Application Case Study: Cargo Container Monitoring

Using the Agimone architecture, how can the cargo container monitoring application mentioned in Section 4.7 be implemented?

As previously mentioned, WSNs can reduce the risk of importing cargo containers by continuously monitoring every container and forming a wireless ad hoc network for delivering alerts. However, developing an application on top of WSNs attached to cargo containers is difficult due to the highly dynamic environment, and the fact that containers tend to be spread around the world.

Agimone facilitates the development of this application by allowing users to inject agents into the network that continuously monitor the sensors for anomalies. Alarms can be raised by inserting a tuple describing the event into both the local tuple space and the base station's tuple space. Using Agimone, the `AgimoneAgent` residing on the base station automatically translates this alert into a tuple that Limone agents understand. Limone agents subscribe to reactions that are senstive to these alert tuples, and notify the user.

To evaluate this application, a mock cargo container test bed was deployed. Each container is given a Mica2 node with a MTS310 sensor board for sensing light, temperature, and vibration, and a speaker for emitting an audible alert. An Agilla agent is implemented that periodically accesses the light sensor to determine whether the container's door is open. If it is, it inserts an alert tuple both locally and in the base station's tuple space. A Limone agent is implemented that reacts to this tuple and displays an alert on the user's display. As a testament to Agimone's expressiveness, the Agilla agent only consists of 17 lines of code, and the Limone agent requires only 11 lines of code. The Agilla agent and the Limone client were developed in only a few hours.

Instead of remotely inserting the the alert tuple into the base station's tuple space, suppose the aforementioned agent only inserts it in the local tuple space. This may be because the container is on a ship in the middle of the ocean where there is no Internet access, or because the alerts are not critical until the ship arrives at a port. In this case, using Agimone, the user can deploy a search agent that traverses a WSN looking for alert tuples and sending them back to the base station. As further testament to Agimone's expressiveness and flexibility, creating an agent exhibiting this search behavior required adding only 23 lines of code to the previous agent. Finally, creating this application spanning both WSNs and IP networks did not require any application-specific integration code.

4.8 Chapter Summary

This chapter presented Agilla, a mobile agent middleware specifically designed for resource-constrained WSNs, and Agimone, a lightweight integration layer that seamlessly merges the Agilla and Limone coordination models, enabling applications to span IP and sensor networks. Agilla is the first working system to bring mobile agents and a tuple space-based coordination model into WSNs. Agilla enhances network flexibility and supports adaptive applications through mobile agents that coordinate via localized tuple spaces. Agilla has been implemented on TinyOS and multiple WSN hardware platforms. Empirical results on Mica2 and TelosB nodes demonstrate the feasibility and efficiency of Agilla on resource-constrained WSNs.

Comparisons with native code-propagation systems indicate that Agilla is comparable to other VM-based systems. Experiences with two adaptive applications, fire tracking and robot guidance, demonstrate the expressiveness of Agilla's programming model. Experiences with a cargo container monitoring application demonstrate the efficacy of Agimone.

Chapter 5

Servilla: Service Provisioning for Wireless Sensor Networks

This chapter focuses on how middleware can address the challenges of programming WSN applications due to network heterogeneity. It is motivated by the fact that WSNs are becoming increasingly heterogeneous consisting of devices with a wide range of capabilities. WSN applications are often optimized for energy efficiency in platform-specific ways, preventing them from executing in heterogeneous networks. The middleware presented in this section, called ***Servilla***, addresses this problem through service provisioning. Using Servilla, developers can construct platform-independent applications over a dynamic and diverse set of devices. The middleware automatically discovers local and remote services that perform platform-specific operations and dynamically binds applications to these services to enable flexible and energy-efficient in-network collaboration among heterogeneous devices. Furthermore, Servilla provides a modular middleware architecture in which parts can be omitted, lowering Servilla's minimum system requirements. This enables applications to operate on powerful devices while leveraging the benefits of resource-constrained devices like density and energy efficiency. Servilla has been implemented on TinyOS for two disparate hardware platforms, the Imote2 and TelosB. Microbenchmarks demonstrate the efficiency of Servilla's implementation, while application case studies involving structural health monitoring demonstrate the efficacy of its coordination model.

5.1 Motivation

There are two primary reasons why WSNs are becoming increasingly heterogeneous. First, heterogeneity allows a network to be both computationally powerful and deployed in high densities. Powerful devices can perform complex computations, but are more expensive and energy inefficient preventing them from being deployed in large numbers. Conversely, low power WSN devices have limited computational capabilities, but are relatively cheap and energy efficient, enabling higher deployment densities and increased network lifetime. By integrating weak and powerful devices, a heterogeneous WSN can combine the best features of each, i.e., high levels of computational power, network densities and lifetimes. Second, network heterogeneity follows from the natural evolution of technological progress. WSN devices are embedded in the environment and remain for long periods of time. During this time, new devices and sensors are developed and deployed, resulting in network heterogeneity.

Network heterogeneity presents a formidable problem for application developers. Traditionally, to maximize energy efficiency and meet severe memory constraints, developers would hand-tune an application for a particular hardware platform. This was possible since most WSNs at the time were homogeneous. With WSNs becoming increasingly heterogeneous, this approach is no longer feasible since the target platform may consist of many different types of devices. Furthermore, the continuous evolution of WSN technologies means that the actual platforms on which an application will execute may not be known at the time the application is written. This suggests that applications should be written in a platform-independent manner. Thus, the challenge is how to ensure that the application remains efficient, despite being platform-independent, while still able to access and fully exploit platform-specific capabilities like sensing. These seemingly contradictory requirements complicate application development and motivate the creation of a new coordination model.

To address the challenges of programming heterogeneous WSNs, we developed Servilla, a middleware supporting a novel coordination model based on *Service-oriented computing* (SOC) [136]. SOC is a programming model consisting of service consumers and providers that are automatically matched and bound by the system, enabling consumers to invoke the services provided. Servilla adopts this programming model by structuring applications as a collection of platform-independent

mobile tasks (consumers) that are dynamically bound to platform-specific services (providers). This is possible due to the decoupling of consumers and providers in the SOC programming model. Servilla exploits this decoupling by enabling tasks to seamlessly invoke bound services in a uniform manner regardless of whether they are local or remote and the type of device that provides them. By facilitating remote service access across heterogeneous devices, extremely resource-poor nodes can still contribute by only providing services and not executing application tasks, thus increasing the range of devices supported by the middleware. This is achieved by modularizing Servilla's middleware architecture, and supporting asymmetric middleware configurations across heterogeneous devices.

Servilla's coordination model tailors SOC to the WSN domain in multiple ways. New forms of service bindings and invocation semantics are introduced that maximize efficiency. For example, in addition to the consumer-initiated on-demand service invocations provided by most traditional SOC middleware on the Internet, Servilla provides two new forms of invocation that are initiated by the provider: periodic and event-based. These forms of invocation facilitate the exploitation of network heterogeneity to increase energy efficiency by allowing high-power nodes to offload continuous but non-computationally-intensive duties like sensor monitoring onto low-power nodes, thus enabling them to remain asleep a larger percentage of the time. In addition, Servilla introduces specialized service specification and task programming languages that capture the novel binding and invocation semantics, while enabling the rapid development of complex applications. Servilla enables, for the first time, in-network collaboration between heterogeneous WSN devices via service provisioning

SOC has long been used on the Internet and has recently been explored in the context of WSNs. Two systems in particular are Tiny Web Services (TWS) [147] and PhyNet^{TM} [9]. TWS implements an HTTP server on each device and enables applications to invoke services using HTTP requests. PhyNet^{TM}provides a central gateway that exposes WSN capabilities as web services. Unlike these systems, which treat the WSN as a data source for applications residing outside of the network, Servilla takes the SOC programming model *inside* a WSN. It exploits the loose coupling between service consumers and providers to separate application-level platform-independent logic from the low-level software components that exploit platform-specific capabilities. Furthermore, by allowing application logic to execute inside a WSN, higher levels

of efficiency are obtainable via in-network processing [78]. For example, in a structural health monitoring application, a low-power device may use a simple threshold-based algorithm to detect potentially damage-inducing shocks, and only activate more powerful devices that perform the complex operations to localize damage when necessary [66]. Or, in a surveillance application, low-power devices may sense vibrations from an intruder and activate more powerful devices with cameras [72]. The ability to support collaboration among heterogeneous devices *inside* a WSN is a key feature that distinguishes this work from other SOC middleware for WSNs.

The remainder of this chapter is organized as follows. Section 5.2 presents related work. Section 5.3 presents Servilla's programming model. Section 5.4 presents Servilla's programming languages. Section 5.5 presents Servilla's middleware architecture and implementation. Section 5.6 presents an empirical evaluation on two representative sensor platforms with diverse resources. Section 5.7 evaluates the efficacy of Servilla by using it to implement a structural health monitoring application. The chapter ends with a summary in section 5.8.

5.2 Related Work

SOC has long been used on the Internet to enable independently-developed applications to interoperate. There are many SOC systems including SLP [86], Jini [92], OSGi [134], CORBA [133], Salutation [31], and Web Services [3]. They provide technologies that enable language-independent communication, which is essential for interoperability. Some of these include SOAP [45], RPC [44], DCOM [45], and WCF [121]. Servilla has three salient features that distinguish it from these SOC frameworks. First, it focuses on how service-provisioning language and middleware can be made extremely lightweight. This is necessary due to the limited resources available on many WSN devices. Many previous SOC systems have been ported to small PDA-class devices, which are still relatively powerful compared to the sensor devices used by Servilla. Second, Servilla is specifically designed for localized service binding which is a common case in WSNs due to limited energy resources. Finally, Servilla provides a modular middleware architecture that can be configured for wireless sensor network platforms that consist of devices with a wider range of resources.

SOC is a topic of interest in the coordination community. For example, new languages have been developed that enable formal reasoning about complex service interactions and compositions [20, 1, 118, 5]. Calculi have been developed to model sessions and multi-party dynamic interactions between service users and providers [120, 28]. New ways of specifying quality-of-service requirements and achieving higher levels of reliability have been proposed [7, 24, 26, 131, 23]. SOC has even been used in non-traditional environments like mobile ad hoc networks [69]. Recently, there has been increased interest in context-aware applications [132, 55, 40]. WSNs, being embedded and able to sense the environment, are inherently context-aware. This paper takes the natural next step of applying SOC principles to WSNs. The key distinguishing feature of Servilla lies in its capability to support both resource-constrained devices and more powerful devices, and its light-weight language and middleware tailored for in-network coordination among sensors.

Efforts to bring SOC technologies into the WSN domain include PhyNet^{TM} [9], Atlas [89], and Tiny Web Services [147]. These platforms optimize Internet protocols to function under the severe resource constraints of WSNs. They differ in that PhyNet and Atlas merely provide the service provisioning interface as a translation layer on the gateway device, whereas Tiny Web Services pushes service provisioning onto each individual sensor network node. Unlike Servilla, they do not allow service consumers to exist *inside* the WSN. Instead, they only enable language-independent communication between services inside the WSN and applications outside of the WSN. Servilla is complementary to these efforts; Servilla may leverage off of these systems to expose WSN services to applications external to the WSN, while these systems may rely on Servilla to bring the full capabilities of SOC inside the WSN itself.

In addition to SOC, Servilla shares the common approach of using scripts in a WSN, though for different reasons. Some scripting systems, including Maté [98], ASVM [100], SwissQM [125], and Agilla, enable reprogramming. Other systems, including Melete [185] and SensorWare [25], enable multiple applications to share a WSN. All of these systems come with different scripting languages [97, 47, 62, 111, 184]. Servilla differs by focusing on challenges due to network heterogeneity and dynamics. Unlike other systems, Servilla allows scripts to remain platform-independent and dynamically find and access platform-specific services. One scripting system, DVM [14], explores the similar idea of integrating platform-independent scripts with

native services. It features a dynamically extensible virtual machine in which services can register extensions. While this enables tuning the boundary between interpreted and native code, DVM does not support flexible matching between scripts and services.

Servilla provides a modular and configurable platform in which extremely resource-poor devices only implement a fraction of the entire framework. This enables a hierarchy in which weak devices serve more powerful devices. The idea of establishing a hierarchy of devices inside a WSN is promoted by other systems. Tenet [60] promotes this idea by creating a two-tired WSN in which the lower tier consists of resource-poor devices that can accept tasks from higher-tier devices. It differs from Servilla in that it does not support flexible service discovery and binding between different devices. SONGS [106] is an architecture for WSNs that allows users to issue queries that are automatically decomposed into graphs of services which are mapped onto actual devices. SONG does not provide flexible service binding among heterogeneous devices. Triage [16] is another system that uses low-power nodes to collect data and determine how to optimally schedule more powerful devices that analyze the data. This differs from Servilla in that the low power nodes manage the high-powered nodes, which is the opposite in Servilla.

5.3 Programming Model

An overview of a WSN using Servilla is shown in Figure 5.1. It consists of devices, tasks, and services. A key feature and novelty of Servilla is its support for service provisioning *inside* the WSN. That is, both the service consumers (tasks), and service providers (devices), reside within the WSN, and all of the responsibilities of service provisioning, like service discovery, matching, and binding, are performed by the WSN nodes. The ability to bring SOC within WSNs required overcoming significant challenges like limited resource availability, but enhances system flexibility due to the decoupling of service consumers and providers, and enables higher degrees of efficiency via in-network processing.

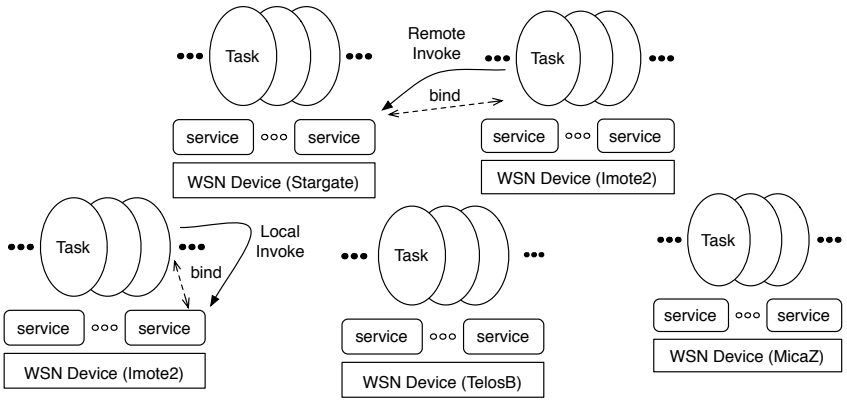


Figure 5.1: Servilla targets heterogeneous WSNs in which different classes of devices provide services that are used by application tasks either locally or remotely. Services are platform-specific while tasks are platform-independent.

Servilla is meant for applications that run in heterogeneous WSNs with multiple classes of devices. It is not intended for flat WSNs composed entirely of resource-poor devices. Typical applications are long-lived, widespread, and involve many different tasks that vary in complexity. They are written once and continuously used despite underlying hardware changes. For example, environmental monitoring and target tracking applications are long-lived and usually involve both simple and complex tasks. Simple tasks, like sensing, are usually widespread and performed by the majority of nodes in the network, while complex tasks, like computations that process the data, are less common and are performed by only a few of the more capable network nodes. By integrating both resource-poor and resource-rich devices, Servilla provides an ideal platform on which to build these types of applications. Specifically, resource-poor devices are less costly and more energy efficient, meaning they can be deployed in greater numbers at higher densities, and can run a larger percentage of the time. Meanwhile, resource-rich devices are more expensive and limited in quantity and energy, but offer computational power and advanced sensing capabilities.

Applications are implemented as tasks, which are platform-independent mobile processes that contain code, state, and service specifications. To ensure platform-independence, the code cannot contain instructions that access platform-specific capabilities like sensors. Instead, these capabilities are accessed as services that are provided by a *service provisioning framework*. The service provisioning framework takes a task's service specifications, finds services that match them, and enables the

task to invoke the service. The service specifications describe both the service's interface and non-functional properties like energy efficiency. This enables a degree of flexibility in selecting a provider. For example, an environmental monitoring application can use the non-functional properties in the service specification to leverage off the most energy-efficient sensors.

Since tasks are platform-independent, they can be mobile, i.e., they can migrate from one node to another, even if the nodes differ in hardware characteristics. This is useful in situations where the network needs to be reprogrammed, or changes in the environment in which the network is deployed result in the need to reconfigure the software executing within the network. For example, the wild fire tracking application presented in Section 4.5 on page 63 needed to reconfigure its software in response to changes in the fire's location. Another scenario in which tasks are mobile is when they execute on mobile nodes. For example, a task belonging to a medical patient monitoring application may execute on a node that is attached to a mobile patient. Task mobility results in the introduction of novel service binding semantics that are described later in this section.

Services expose platform-specific capabilities, are implemented natively, and can thus be fine-tuned for maximum efficiency. They provide a description that can be compared with the service specifications provided by a task. When a match exists, the service fulfills a task's requirement and can be used by the task to perform platform-specific operations. Services are able to maintain state, provide multiple methods, and have their own thread of control, enabling them to operate in parallel with tasks. This enables higher degrees of concurrency and efficiency. For example, in a structural health monitoring application, a service provided by a low-power device can continuously monitor an accelerometer and set a flag if the vibrations exceed a threshold. Using this service, a task executing on a more powerful device can remain asleep most of the time, only periodically waking to check for potential damage.

The mechanism by which tasks communicate are not shown in Figure 5.1 because service provisioning is the focus and main contribution of this work. Tasks communicate via localized tuple spaces [57] that are structured in the same manner as in Agilla. Specifically, each node maintains a separate tuple space that is accessible to

tasks that reside both locally and remotely. Tuple space coordination facilitates decoupled communication, allowing better adaptation to a changing network and task mobility. They serve as a flexible means of communication between application tasks and are orthogonal to service provisioning. While service provisioning messages could be sent using tuple spaces, they are sent in an RPC-like [44] fashion in the current implementation. Regardless of the underlying communication mechanism used by the service provisioning framework, the application is presented with a simple interface for binding to and invoking services, which is presented in Section 5.4.

Tasks remain platform-independent by delegating all platform-specific operations to services. There are two essential steps for this to occur: *binding* and *invocation*. Binding is the process of discovering and establishing a connection to the service. Invocation is the process of accessing a service. The remainder of this section discusses each of these steps.

5.3.1 Service Binding

Service binding consists of three-steps: discovery, matching, and selection. Discovery involves finding available services. In many traditional SOC frameworks, this is done by querying a central service registry. While this is sufficient in traditional networks, it is not appropriate in WSNs. First, since most WSN devices operate on batteries, accessing a distant registry is not energy efficient and can unacceptably reduce network lifetime. Second, the spatial aspects of WSNs are relevant since closer services are usually preferred, e.g., if a task wants to know the temperature, it usually wants to know the local temperature rather than a distant location's temperature. Third, WSNs are ad hoc meaning wireless links are transient and opportunistically formed. Thus, maintaining a route to a centralized registry may be difficult, if not impossible, due to unreliable connectivity. For these reasons, Servilla is optimized for *localized* coordination and does not rely on a centralized service registry. Instead, each device has its own registry containing only a localized view of the services available.

During the service discovery process, the local registry is first checked for a match. If no match is found, neighboring devices are checked. This increases a network's flexibility by allowing tasks to run on devices that do not fully satisfy the service

requirements, since missing services can be provided by neighboring devices. Furthermore, although accessing a remote service requires wireless communication, energy efficiency can be increased overall by allowing high-power devices to use low-power ones, enabling the devices that are less energy efficient to remain asleep longer.

Service matching involves finding a service that fulfills a task's requirements. Recall that tasks include specifications that can be compared to descriptions provided by services. The matching process must be flexible since the service and tasks are usually developed separately. Yet, it must be semantically correct to ensure that the service behaves in a predictable manner. A service is minimally described by its interface. Ideally, the names of the methods, the order, number, and types of their parameters, and even the return types should not require an exact match for service binding for maximum flexibility. To achieve this, large amounts of meta-data must be included in the specification to describe the method names, input parameters, and return values. Unfortunately, such a specification is verbose and requires a complex parser, both of which consume sizable computational resources like memory that are not available on many WSN devices. To account for this, Servilla compromises by dividing specifications into functional and non-functional properties. Functional properties include the interface and require an exact match. Nonfunctional properties describe attributes like power consumption and do not require an exact match. For example, suppose a FFT-calculating service has a non-functional attribute specifying that it is version 5. Such a service can be bound to a task that specifies it requires *at least* version 4. By enforcing an exact match between functional properties and an inexact match between non-functional ones, Servilla provides a degree of flexibility when binding services while still maintaining correct matching semantics and reasonable resource requirements.

Once a matching service is found, the binding process is completed by selecting it. Selection consists of informing the task of the chosen service, and is accomplished by saving the provider's network address in the task's state. Once saved, the task is able to access the service by invoking it. Note that this address is hidden from the application developer, who is able to invoke the service based on its name, a process that is described later in this section.

	Eager	**Lazy**
Persistent	Immediate and frequent invocation of a particular service	Eventual frequent invocation of a particular service without initially knowing which
Transient	Immediate but infrequent invocation of any matching service	Eventual infrequent invocation of any matching service

Table 5.1: Various service binding semantics and when they should be used.

5.3.2 Novel Binding Semantics

Servilla tailors the SOC programming model to WSNs by introducing new binding semantics. This is necessary because of the resource scarcity and dynamics present in most WSNs. Specifically, service bindings may be *eager* or *lazy*, and *persistent* or *transient*. Each of the four combinations specify different binding semantics and are useful in different scenarios that are summarized in Table 5.1. The eager/lazy attribute controls how quickly Servilla performs service discovery after the task issues a bind request. If eager, the discovery process is initiated immediately. If lazy, the process is initiated upon first invocation. Eager binding is faster but may result in needless service discovery if the service is not invoked for a long period of time, especially if the wireless link to the service provider breaks between the time of service discovery and invocation. Lazy binding does not incur any overhead until the service is first invoked, but at the cost of higher initial latency due to the need to perform service discovery.

In most circumstances, eager binding is used since an application usually needs to invoke a service immediately after binding to it. However, in some situations, lazy binding is preferred. For example, suppose a script binds to a service, but then migrates onto a different node before it first invokes the service. In this case, eager binding may result in a suboptimal service being selected since it was chosen based on the original location of the task. Lazy binding is preferred in this situation since the service will not be discovered and bound until the script has arrived at the new node and actually needs to invoke the service.

The persistent/transient attribute specifies what happens after a service is invoked or a task migrates. If persistent, the service remains bound until it is broken due to

network disconnection, node unavailability, or an explicit unbind operation executed by the task. If transient, the service is unbound after being invoked or the task moves. Persistent bindings enable tasks to invoke the same service multiple times and are especially useful if the service must be invoked frequently since they do not require that the service be rediscovered and rebound prior to each invocation. In contrast, transient bindings are unbound after each invocation, freeing memory resources. They are useful for one-time or infrequent invocations.

5.3.3 Service Invocation

Service invocations are the means by which services are executed. Servilla offers three forms of service invocation: *on-demand*, *periodic* and *event-based*. On-demand invocations are the simplest and are analogous to remote procedure calls (RPCs) [44]. They are the form offered by most traditional SOC systems that operate on the Internet. To perform an on-demand service invocation, the task sends a message to the provider containing the specification of which service to invoke and any necessary input parameters. Upon receiving this message, the provider executes the service and sends a reply containing the results of the invocation to the task. This form of service invocation must be re-issued by the task each time it requires the service to be executed. Upon receiving the invocation request, the service is immediately executed. The service is not executed unless a service invocation request is received, thus the name "on-demand."

The other two forms of service invocation, *periodic* and *event-based*, are motivated by the fact that on-demand invocations can be optimized in certain situations common to WSN applications. Specifically, many WSN applications like habitat monitoring require the same service to be repeatedly invoked using the same set of parameters for extended periods of time. In this situation, requiring the task to resend the message initiating the invocation each time it needs the service to be executed, as is the case with on-demand invocations, is not energy-efficient. For this reason, Servilla introduces periodic and event-based forms of service invocation. These two forms of service invocation enable the provider to automatically execute the service for a consumer. This is done by requiring the task to specify the period at which it needs the service executed. The provider will then execute the task at the specified period. Periodic

and event-based invocations differ in terms of results delivery. Periodic invocations send every invocation result back to the consumer whereas event-based invocations only send "interesting" results. For example, a service that monitors the accelerometer may consider the results of an invocation interesting if the recent acceleration readings exceed a certain threshold. In the current model, the service itself determines whether the results of invoking it is interesting. An alternative is to enable the consumer to specify which results it considers interesting, perhaps through predefined parameters or the use of mobile code. Both forms of invocations are more energy efficient since they do not require the consumer to send the provider a message each time the service is invoked. Event-based service invocations are even more efficient than periodic invocations since they eliminate needlessly sending uninteresting results back to the consumer task.

Since the task and service may be located on different devices, the service invocation process may fail, e.g., due to message loss. To account for this, Servilla provides a mechanism that notifies a task when and why an invocation fails. This is necessary because service invocations may fail in many ways depending on whether the service is local or remote, and tasks may want to handle various error conditions differently. For example, local invocations may fail because the service is busy, in which case the task may try again later, while remote invocations may fail due to disconnection, in which case the task may want to abort and switch to a different provider.

5.4 Programming Languages

Servilla provides two light-weight programming languages tailored to support service provisioning in WSNs. The first, *ServillaSpec*, is used to create service specifications and descriptions that enable flexible matching between tasks and services. The second, *ServillaScript*, is used to create tasks and is compiled into bytecode that runs on a virtual machine, which is used to ensure platform-independence. Services are implemented in NesC [56] on TinyOS [73] and compiled into native binary code for runtime efficiency. Each of Servilla's specialized languages are now described.

NAME = fft
METHOD = fft-real
INPUT = {int dir, int numSamples, float[] data}
OUTPUT = float[]
ATTRIBUTE Version = 5.0
ATTRIBUTE MaxSamples = 5000
ATTRIBUTE Power = 10

Figure 5.2: A specification describing a FFT service

5.4.1 ServillaSpec

The simplest language encapsulates the same amount of information as a typical interface class provided by object-oriented programming languages like Java or C++. The specification must include the name of the service followed by a sequence of methods, where each method is specified by its name, input parameters, and return type. While this is sufficient for establishing a compatible match, it requires an *exact match* between the specifications provided by the service and task. This lack of decoupling can lead to unexpected mismatches since services and tasks are usually developed separately. A more flexible language that enables specifications to match in an inexact manner is necessary to ensure interoperability between tasks and services.

On the opposite side of the flexibility spectrum is a very expressive language like that provided by service specification languages used on the Internet. One example language is the Web Services Description Language (WSDL) [171]. WSDL is written in XML using vocabulary specified by an XML schema [172]. This combination is often used to provide web services over the Internet. Using this language, a service specification may contain as much metadata as necessary to fully describe the service. This enables extremely flexible matching between tasks and services, but requires a complex interpreter that consumes more memory than is available on WSN devices [147]. For example, WSDLInterpreter [87] is a program that interprets a WSDL document. Its code is 48KB, which is the *total* amount of code memory on the TelosB [144] platform.

ServillaSpec is used to describe services in Servilla and is needed to match services required by tasks to those provided by devices. To support resource-constrained devices, the service specification language must be compact and should not require

an overly complex matching algorithm. It attempts to strike a balance between service matching flexibility and overhead. ServillaSpec avoids verbose syntax and limits the types of properties that can be included in a service specification. An example is shown in Figure 5.2. The first line specifies the name of the service. It is followed by three-line segments each specifying the name, input parameters, and output results of a method provided by the service. The remainder of the specification is a list of attributes that specify non-functional properties of the service. They enable flexibility in matching by defining a name, relation, and value. Possible relations include <, >, <=, >=, and =. Using attributes, a task can, for example, require a floating point FFT service that consumes *at most* 50mW. Such a specification would match a service whose description is shown in Figure 5.2.

ServillaSpec achieves flexibility in two ways. First, the number of attributes and methods provided by the service need not match the number provided by the task. A match will occur so long as each attribute and method specified by the task is satisfied by an attribute or method provided by the service. That is, the task specification can be satisfied by a subset of the service's specification. A service may have more attributes and methods than is necessary for a match to occur. Second, attributes contain relations that enable inexact matches. For example, the application developer may specify an attribute with a minimal value, while a matching service may have a value that is greater than this minimum. The above two characteristics enable flexible matching between application tasks and services that have non-identical specifications, which is essential in a dynamic and heterogeneous WSN.

Servilla relies on a globally-defined vocabulary that specifies the meaning of each attribute. This ensures no confusion regarding, for example, the meaning of "MaxSamples" in the specification shown in Figure 5.2. Servilla also requires that there be a globally-defined manual that specifies the meaning of each unique service signature, which consists of the name and method properties. For example, by looking up the signature of the specification defined in Figure 5.2, the programmer will learn three things: 1) The service performs a FFT and one of its methods does the FFT on an array of real values, 2) The input parameters specify the FFT direction, number of samples, and the samples on which to perform the FFT, respectively, and 3) the output is the result of the FFT. If the service requires or outputs data with units,

NAME = Accel
METHOD = readx
INPUT =
OUTPUT = int
ATTRIBUTE Latency = 5
ATTRIBUTE Error = 2

Figure 5.3: Possible ambiguity: Does attribute Error modify the Latency attribute or the readx output?

the manual must also include the unit specifications. For example, for a temperature sensing service, the manual must specify whether the output is in Fahrenheit or Celsius. The service manual is necessary for developers to understand the semantics of the specification's signature and, ultimately, whether the service that advertises such a specification meets the needs of the application. Once the developer decides to use a certain specification, the matching between service specifications and script specifications is done automatically.

By limiting the property types to be only the five shown in Figure 5.2 (i.e., NAME, METHOD, INPUT, OUTPUT, and ATTRIBUTE), and arranging them to always be in the same order, the specification can be greatly compressed. For example, since the service's NAME property always appears first, the property's identifier, NAME, can be omitted. Thus, the NAME property in the specification shown in Figure 5.2 can be compressed to just 4 bytes, "fft" followed by a null terminator. This compression saves memory and enables greater matching efficiency.

When creating a service specification, care must be taken when including attributes to prevent ambiguity. For example, consider the service specification shown in Figure 5.3. The specification describes a service that provides an acceleration reading along the x-axis. There are two attributes, Latency and Error. The ambiguity arises regarding whether the Error attribute modifies the latency, or the output of method readx. To prevent this ambiguity, the name of the attribute must clearly indicate what is being modified. For example, instead of "Error," it should be renamed to "Error-readx" to indicate that it specifies the maximum error of the acceleration measurement.

import fft-v5
ATTRIBUTE Version = 6.0

Figure 5.4: ServiceSpec specifications are extensible using the import keyword

Service specifications must be extensible to adapt to new services that become available. A specification can inherit the properties of an existing specification using keyword **import**, and override any of the inherited properties. For example, suppose a new FFT algorithm is implemented. A service that provides this new FFT algorithm can advertise the specification shown in Figure 5.4. Assuming the specification shown in Figure 5.2 is saved in a file called "fft-v5," this new specification will be identical except its version attribute will have a value of 6. Note that *all* properties including the optional attributes are inherited. This ensures that all children specifications are at least as well specified as the parent.

While the ServillaSpec language does provide a mechanism for achieving a degree of flexibility when determining a match between two specifications, in the form of attributes, the supporting middleware can enhance this flexibility by providing simple translation services. For example, suppose a task requires the temperature in Fahrenheit but the only temperature sensing services in range provide the temperature in Celsius. If the middleware could provide a translation service that automatically converts the output of the services to Fahrenheit, a match could be established. Another possibility is for the middleware to automatically build composite services that fulfill a task's requirement where each individual service does not. An example is a task that requires a light sensing service that covers both the visible and invisible light spectrum, but the only services that are available provide one or the other. There is a large body of work related to service composition [145]. These efforts can be incorporated into the Servilla middleware framework to enhance the flexibility of service matching.

5.4.2 ServillaScript

ServillaScript is the language used to create application tasks. Its syntax is similar to other high level languages like JavaScript [51], but with key extensions for service

```
1.  // Declare which services are required.
2.  uses Accel;
3.
4.  // Begin task execution.
5.  void main() {
6.      int count = 0; float accel;
7.      bind(Accel, EAGER|PERSISTENT, 2); // bind to a service within 2 hops
8.      while(count++ < 10) {
9.          accel = invoke(Accel, "readx"); // invoke the service
10.         send(accel);
11.     }
12.     unbind(Accel);
13. }
```

Figure 5.5: A task that invokes an accelerometer-sensing service 10 times

```
1.  uses Accel as Accel1;
2.  uses Accel as Accel2;
3.  ...
```

Figure 5.6: The **uses** keyword allows a script to use multiple services that have the same specification.

provisioning. An example, shown Figure 5.5, implements an application that periodically takes the acceleration reading and sends the reading to the base station. It declares the name of the file containing the specification of the required service on line 2, which in this case is an accelerometer-sensing service. The task initiates the service binding process on line 7, and then invokes the service ten times (line 9), each time sending the results of the accelerometer reading to the base station (line 10). The task ends by unbinding from the service on line 12.

All tasks begin with a declaration of which services are required. This is done using the **uses** keyword, which specifies the names of the files containing the specifications of the required services. The same name is also used in the body of the task to refer to the service during the binding, invoking, and unbinding operations. One limitation of the **uses** syntax, as currently described, is that a task can only bind to one of each type of service. To address this, a task can use the **as** keyword to rename a service specification allowing it to bind to multiple services with the same specification. For example, if a task requires two accelerometer-sensing services, it can use the code shown in Figure 5.6. The body of the task can then bind to

and invoke two accelerometer-sensing services, which are referred to as Accel1 and Accel2, separately. Note that there is no guarantee that the middleware will select two physically different services. In the example above, it is possible for Temp1 and Temp2 to be bound to the same physical sensor. However, the task may use each service differently in terms of invocation times and binding semantics. The syntax for specifying the binding semantics is discussed next.

An example of the binding syntax is shown on line 7 of Figure 5.5. It consists of the keyword **bind** followed by three parameters: the name of the service to bind, the binding semantics, and the number of hops to search. The first parameter corresponds to the name of the service, which was previously declared by the **uses** keyword. The second parameter specifies whether the binding should be eager/lazy and persistent/transient. The last parameter is an integer that controls how far the service provider can be relative to the task. This is important to control the overhead of service invocation. For example, if the task insists on a local service, it can set this value to be zero.

The bind operation is performed synchronously with the task. That is, the task is blocked while the middleware attempts to find a matching service provider, and can only resume execution when a provider is found, or the service discovery process fails to find a matching provider and aborts. To check whether a service was successfully bound, the task can call **isBound(service name)**, which returns a boolean value indicating the success of the bind operation. For example, line 7 of Figure 5.5 can be followed by **if**(**isBound**(Accel)){ ...} to double check the success of the bind operation before it starts to invoke the service. Another command the task can execute is **numHops(service name)**. This command returns the number of hops away the service is relative to the task, and allows the task to control the overhead of service invocation. For example, the task can throttle the service invocation frequency based on the network distance to the service.

The actual service invocation is done using one of three keywords, **invoke**, **invokePeriodic** and **invokeEvent**, depending on the type of invocation being performed. The **invoke** keyword performs on-demand service invocation, and an example of its use is shown on line 9 of Figure 5.5. Its first parameter specifies the name of the service being invoked, and the second parameter specifies the method within the service to

```
1. uses Accel;
2.
3. void main() {
4.     int success;
5.     success = invokePeriodic(Accel, "readx", 1024, gotAccel);
6.     if (success) { ...}
7. }
8.
9. void gotAccel(int reading) { ...}
```

Figure 5.7: The **error** keyword specifies an error callback function that is executed when the invocation fails.

execute. If the service method requires input parameters, they would be in included after the second parameter of the **invoke** command. Like **bind**, **invoke** is performed synchronously meaning the task blocks until the invocation completes or fails. The results of the invocation are returned by the **invoke** command itself.

The **invokePeriodic** and **invokeEvent** commands differ slightly from **invoke** in that they are performed asynchronously with the task and use a callback function to deliver the results of the service invocation. An example of how **invokePeriodic** is used is shown on line 5 of Figure 5.7. The **invokeEvent** command is used in a similar fashion as **invokePeriodic**. Like **invoke**, the first parameter of **invokePeriodic** specifies the name of the service being invoked, and the second parameter specifies the name of the method within the service to execute. In addition, **invokePeriodic** takes two more inputs: the period at which the service should be invoked and the name of the callback function that should be called each time the results of the service invocation are received. In the example shown in Figure 5.7, the Accel service is invoked periodically every second, and the gotAccel(int) method is called each time the service is invoked. The execution of gotAccel(int) is analogous to that of an interrupt. Specifically, it forces the task to pause its current execution, run the invocation callback function to completion, and then resume where it left off. Note that the parameters of the callback function must match the output of the method as specified by its service specification. In this case, the parameter consists of a single integer, which matches the output of method "readx" in the specification shown in Figure 5.3. The return value of **invokePeriodic** and **invokeEvent** indicates whether the invocation was successfully started.

```
1. uses Accel;
2.
3. void main() {
4.     invoke(Accel, "get") error invokeFailed;
5.     // ...
6. }
7.
8. void invokeFailed() { ...}
```

Figure 5.8: The **error** keyword specifies an error callback function that is executed when the invocation fails.

The dynamic nature of WSNs result in the possibility that service invocations fail. This is due to the wireless disconnection between the task and the previously-bound service. To handle these situations, ServillaScript provides the **error** keyword that can be included in the **invoke** command. An example of its use is shown on line 4 of Figure 5.8. The **error** keyword specifies a callback function that should be called in case the service invocation fails. In the example shown in Figure 5.8, the error callback function is invokeFailed. It can also be included in the same manner with the **invokePeriodic** and **invokeEvent** commands. After the callback function executes, the script returns to the line following the invocation command. The error keyword enables the application developer to account for situations where a service provider unexpectedly disconnects during the service invocation process.

This concludes the discussion of Servilla's programming languages. Servilla provides two specialized programming languages, ServillaSpec and ServillaScript, that enable the creation of service specifications and application tasks, respectively. ServillaSpec enables flexible yet efficient matching between services and tasks. ServillaScript enables platform-independent applications to be developed that rely on available services to exploit platform-specific functionality and achieve high levels of efficiency. The next section discusses Servilla's middleware.

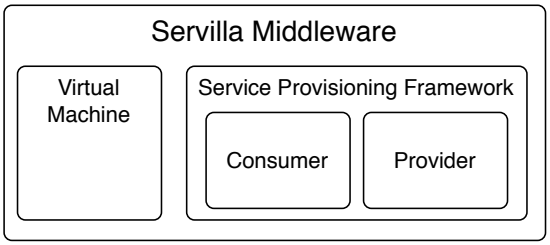


Figure 5.9: Servilla's middleware consists of a virtual machine and a service provisioning framework (SPF). The SPF consists of a consumer and provider.

5.5 Middleware Architecture and Implementation

Servilla's middleware architecture is shown in Figure 5.9. It consists of a virtual machine (VM) and a service provisioning framework (SPF). The VM is responsible for executing application tasks. The SPF consists of a consumer (SPF-Consumer) that discovers and accesses services, and provider (SPF-Provider) that advertises and executes services. A VM is used because WSN devices contain processors that have non-uniform instruction sets. If a task were compiled for one WSN device, it may not be able to execute on another device with a different instruction set, violating the premise that tasks be platform-independent. Instead, tasks are compiled into the VM's instruction set, which is uniform across all hardware platforms, ensuring that tasks are platform-independent. In addition, the VM facilitates the dynamic deployment and mobility of tasks, further motivating the need for dynamic service binding and the novel binding semantics. Any number of VMs for WSNs can be used [14, 98, 100, 125, 185], so long as they can be extended to support services and the SPF. Specifically, whenever a task performs an operation involving a service, the VM passes the task to the SPF-Consumer, which is described next.

5.5.1 SPF-Consumer

The SPF-Consumer is responsible for discovering, matching, and invoking services on behalf of tasks. As shown in Figure 5.10, the SPF-Consumer consists of a Service Finder, Binding Table, and Service Scheduler. The Service Finder is responsible for finding services that match a task's specifications, and enforcing the binding semantics

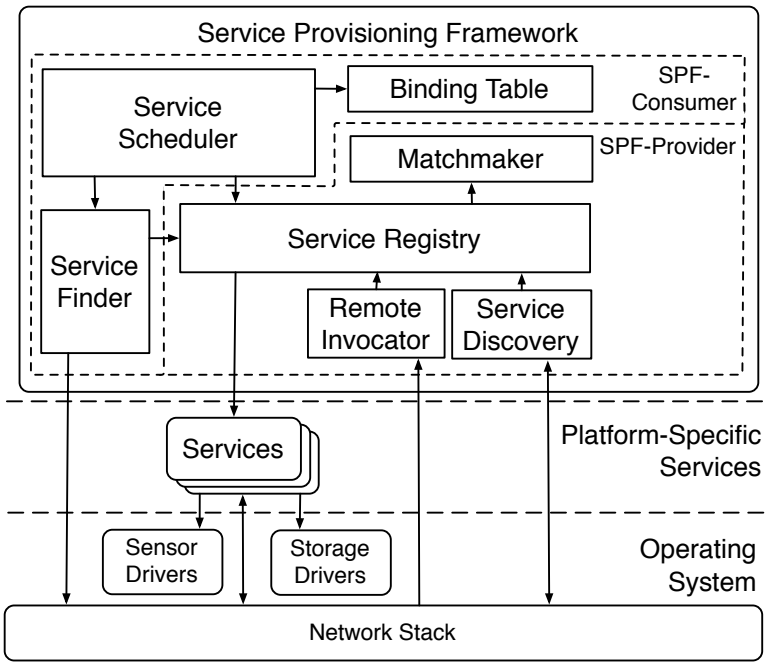


Figure 5.10: The detailed architecture of the Service Provisioning Framework.

specified by the application task. It first searches locally and, if no matches are found, searches neighboring devices. Note that while this increases the likelihood of selecting a local service, it does not necessarily select the service that is most energy-efficient. If a task wanted to consider energy-efficiency in the service selection process, it can include an energy attribute in its service specification, forcing the SPF-Consumer to select a service that meets the energy-efficiency specification. Through this mechanism, energy-efficient service provisioning can be achieved.

When a provider is selected, its address is stored in the Binding Table. The Binding Table maps the task's service specification to the provider that will perform the service. It is updated when the Service Finder discovers a new provider and when a task explicitly unbinds from a service. A task can query the Binding Table to determine whether it has access to a particular service.

The Service Scheduler carries out the actual invocation. It takes the invocation specifications (e.g., type of invocation) and input parameters provided by the task, sends them to the provider, and waits for the results to arrive. Once the results arrive, it passes them to the task, which can then process the results. If the results

do not arrive within a certain time, the Service Scheduler aborts the operation and notifies the task of the error. In the case of periodic invocations, the Service Scheduler monitors the periodic reception of invocation results, and alerts the task if a failure has occurred. For event-based invocations, the Service Scheduler monitors the continued presence of the provider, and notifies the task if the provider is disconnected while the service is still being invoked.

5.5.2 SPF-Provider

The SPF-Provider is responsible for providing and executing services. Its architecture, shown in Figure 5.10, consists of a Service Registry, Matchmaker, Remote Invocator, and Service Discovery component. The Service Registry contains the specifications of all locally-provided services. It may also cache the specifications of services provided by neighboring nodes that have recently been discovered. This reduces the overhead of remote service discovery by increasing the likelihood that a matching service be found in the local Service Registry. Note that each Service Registry is independent and contains a different set of services based on what is locally available. This is necessary to limit the overhead of storing and maintaining the Service Registry.

The Matchmaker is used to determine whether a service meets the task's requirements. When the SPF-Consumer tries to find a service, the Matchmaker takes the specification provided by the task, and compares it to the specification provided by each service in the Service Registry. If it finds a matching specification, the Matchmaker returns a positive response. Note that in this architecture, the task's specification must be sent from the SPF-Consumer to the SPF-Provider since that is where the Matchmaker is located. Alternatively, the Matchmaker can be moved onto the SPF-Consumer to reduce the footprint of the SPF-Provider. However, this requires all specifications belonging to all services to be sent to the SPF-Consumer for service matching to be performed, a process that may incur higher communication cost since, in most situations, there are more services provided than required by a task. Assuming the Matchmaker is moved onto the SPF-Consumer on some devices, it is possible to encounter a situation in which neither the SPF-Consumer nor SPF-Provider implement the Matchmaker. In this case, no service matching can occur and the services provided by the SPF-Provider are not considered in the service discovery process. To

minimize the occurrence of this scenario, the Matchmaker should be implemented on the SPF-Provider whenever possible.

When a script invokes a remote service, the SPF-Consumer sends the input parameters to the SPF-Provider on the device that provides the service. The Remote Invocator component within the SPF-Provider receives the input parameters and passes them to the Service Registry, which executes the service. In the case of periodic and event-based invocations, the Remote Invocator performs the periodic execution of the service as specified by the task. Note that the services shown in Figure 5.10 access platform-specific functions like sensor and storage drivers, as well as the network stack for providing services that require network communication.

5.5.3 Middleware Modularity

WSNs are becoming increasingly diverse consisting of devices with resources that differ by several orders of magnitude [144, 37]. This will remain true even as technology improves, since cost considerations ensure the continued presence of resource-limited devices. To accommodate the wide range of devices, Servilla's middleware is modularized and configurable such that a device need not implement every module to participate in the network. For example, the following are three ways in which the middleware can be configured. This list is not complete, but rather contain the most commonly used middleware configurations.

- **VM + SPF**. This configuration provides the entire Servilla framework and can only exist on relatively resource-rich devices like the Imote2 [37] and certain configurations of the mPlatform [110]. It allows application tasks to execute on the device and invoke both local and remote services.
- **VM + SPF-Consumer.** In this configuration, an application task can execute on a device, but only invoke remote services because a SPF-Provider is not present locally. It frees up a significant amount of resources since it eliminates half of the SPF and, more significantly, the services that may require complex

	TelosB	Imote2
Processor	8MHz 16-bit TI MSP430	13-416MHz 32-bit Intel PXA271 XScale
Radio	IEEE 802.15.4	IEEE 802.15.4
Memory	48KB Code, 10KB Data	32MB Shared
Price	$99	$299

Table 5.2: WSN devices vary widely in computational resources.

drivers for accessing platform-specific hardware. As mentioned previously, depending on resource availability, the SPF-Consumer may or may not implement the Matchmaker.

- **SPF-Provider Only.** This configuration is especially useful for severely resource-poor devices. While it cannot host application tasks, it can dedicate all of its resources to providing services. Scripts residing on remote devices may then invoke these services, enabling even resource-poor devices to participate. The smallest Servilla configuration is this configuration without a Matchmaker.

A detailed analysis of the memory consumed by each configuration is given in Section 5.6.1. The configuration containing only the SPF-Provider is particularly interesting because it allows resource-weak but energy-efficient devices to provide services to more powerful devices. This can result in greater overall energy efficiency and, assuming the weak devices are less costly and more numerous, increase sensing density while achieving greater sensing coverage. The various middleware configurations are *transparent* to tasks due to the decoupled nature of the SOC model. For example, a task need not know whether there is a local SPF-Provider. If a task requires a service, it will be bound either locally or remotely depending on availability.

Servilla has been implemented on TinyOS 1.0 and two representative hardware platforms shown in Table 5.2. It is divided into two levels as shown in Figure 5.11: a lower level consisting of shared components and a higher level consisting of Servilla's VM and SPF. This section first discusses the lower level followed by the upper level. It ends with a discussion of Servilla's programming languages.

The shared components implement low-level mechanisms needed by most high-level components. The dynamic memory manager makes more efficient use of memory. This is important because Servilla has several components that require varying

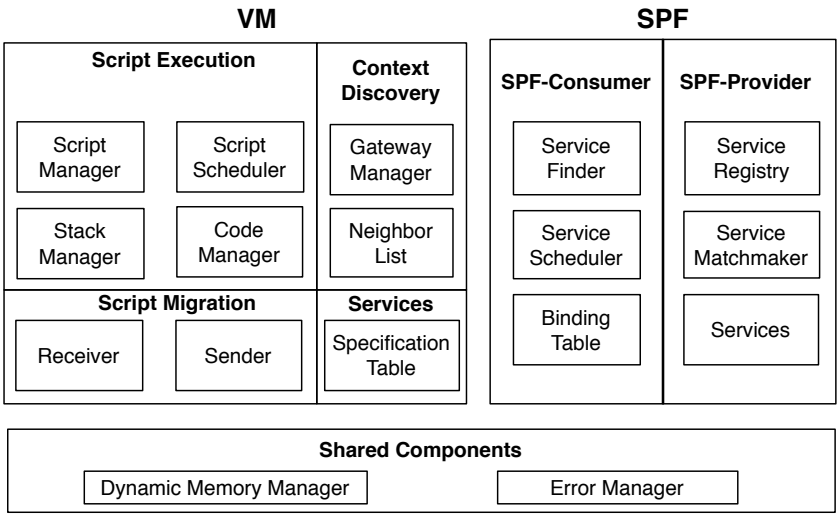


Figure 5.11: Servilla's middleware components.

amounts of memory over time, and TinyOS does not support dynamic memory. The dynamic memory manager provides just enough memory for each higher-level component to complete their function and reclaims the memory when it is no longer needed. It is shared by most components in Servilla's middleware, maximizing the flexibility of memory allocation. To aid in debugging, Servilla provides an error manager that detects and sends summaries of problems to the base station. The error manager is shared by all other components in Servilla's middleware.

Servilla's VM is based on the one provided by Agilla but modified to support the SPF. Its components are shown in Figure 5.11. Unlike most WSN VMs, Agilla provides a particularly powerful *mobile agent* abstraction in which application tasks are able to explicitly migrate across nodes while maintaining their state. Applications whose structures are static, evolving, and even mobile can be designed with equal ease and can coexist on the same WSN. The Agilla VM is modified by augmenting it with a service specification table and service provisioning instructions. When a task performs an operation involving a service, the VM passes the task to the SPF-Consumer.

The SPF is implemented natively using NesC and is divided into two modules, the SPF-Consumer and SPF-Provider, as shown in Figure 5.11. In the SPF-Consumer, the implementation of the Service Scheduler is simplified by serializing service invocations. This has the added benefit of avoiding saturation of the wireless channel. To

```
interface Service {
    command result_t isAvailable();
    command void getSpec(uint8_t** spec, uint32_t* size);
    command result_t invoke(Script* s);
    event result_t done(Script* s, result_t result, uint8_t isInteresting);
}
```

Figure 5.12: All services must provide this interface.

increase energy efficiency, the Service Finder first searches the local Service Registry, if one exists, before searching those of one-hop neighbors. Currently, only one-hop neighbors are supported, the implementation can be extended to support multi-hop service discovery and invocation.

In the SPF-Provider, the Service Registry maintains a list of local services by exploiting TinyOS' ability to parameterize interfaces. Every service provides at least one instance of interface `Service`, which is shown in Figure 5.12. This is wired to the Service Registry using an 8-bit parameter, meaning each node can support up to 256 local services. Currently, the Service Registry only records local services. Remote services are discovered on-demand and must be re-discovered each time they are bound. An enhanced implementation of the Service Registry would include a cache for storing remote specifications to reduce wireless transmissions and service discovery latency.

Servilla's compiler translates tasks and service specifications written in ServillaScript and ServillaSpec into a compact binary format. The compilers are implemented using a scanner created by JLex [49] and a parser created by CUP [158]. The parse tree created by CUP is used to generate the binary encoding of scripts and specifications. Servilla's compilers are able create compact code. For example, the task shown in Figure 5.5 is compiled into 181 bytes of code and 30 bytes of specifications, and the specification shown in Figure 5.2 is compiled into just 64 bytes. Both the Servilla middleware and compiler have been released as open-source software available at `http://mobilab.wustl.edu/projects/servilla/`.

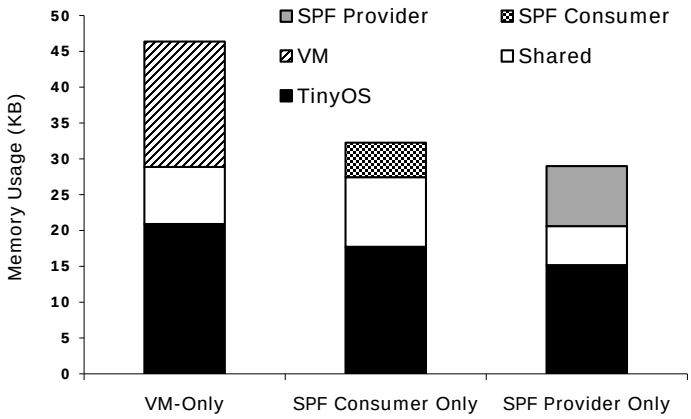


Figure 5.13: The code memory footprint of different Servilla configurations on the TelosB platform.

5.6 Micro-benchmarks

This section presents empirical measurements of Servilla's code size and performance overhead on the TelosB [144] and Imote2 [37] platforms, which vary widely as shown in Table 5.2. The evaluation consists of two parts. First, the memory footprint of the middleware is measured. This determines how well the middleware accommodates nodes with varying amounts of memory. Second, the efficiency of service discovery and invocation is evaluated. The efficacy of the Servilla programming model is demonstrated through an application case study in the next section.

5.6.1 Memory Footprint

The modularity of Servilla's middleware architecture enables its memory footprint to be adjustable. This is important because of the large variation in the amount of memory available on WSN nodes. For example, an Imote2 has 32MB of memory, which is sufficient to hold the entire Servilla middleware. Compiled for the Imote2, the total size without services is a mere 318KB. This is only about 1% of the total, leaving plenty of memory for services. In contrast, TelosB devices only have 48KB of code memory and 10KB of data memory. This is not enough to hold the entire Servilla middleware, as shown in Figure 5.13. The figure shows the amount of code memory consumed by three different configurations of Servilla on the TelosB platform. The

first configuration illustrates the amount of memory consumed on the TelosB when just the VM is installed. In this case, it consumes 45KB on the TelosB. The second configuration consists of just the SPF-Consumer. While this is an invalid configuration (since no tasks can make use of the SPF-Consumer), it is given to illustrate why TelosB nodes cannot include both the VM and the SPF-Consumer. Specifically, the SPF-Consumer consumes 32KB, of which 5KB is unique. This additional 5KB on top of the 46KB consumed by the VM exceeds the 48KB available on the TelosB node, preventing the TelosB from running both the VM and SPF-Consumer. Although TelosB devices do not have enough memory to hold the entire Servilla middleware, it *can* support a configuration consisting of just the SPF-Provider, which only consumes 32KB of code memory as shown by the right-most configuration in Figure 5.13. This example shows how Servilla's modular architecture enables support of diverse hardware platforms.

By allowing computationally weak devices like the TelosB to join and contribute to a WSN as service providers, they can be exploited by more powerful devices. This is important because weaker devices tend to have higher energy efficiency and lower monetary cost, which is the case with the TelosB relative to the Imote2. As shown in previous work [60] and our case study presented in Section 5.7, effective integration of resource-constrained and resource-rich devices can combine the advantages of pervasive low-power sensing with high computational ability, enabling complex applications with enhanced energy efficiency.

5.6.2 Efficiency of Service Binding

Service binding consists of three parts: discovery, matching, and selection. This study first focuses on discovery followed by matching and selection. Recall that, in the current implementation, the Service Finder queries each neighbor individually for a match. This is because the delivery of the service specifications used in determining a match must be reliable, and the reliable network interface that Servilla uses does not support wireless broadcasts. To optimize the selection, the Service Finder first searches locally before remotely. Since the latency of a local search is negligible, we evaluate the latency of a remote search.

Property	Size (Bytes)
Signature	23
Attr. 1	14
Attr. 2	15
Attr. 3	12

Table 5.3: The size of the properties within service specification `FFT`

device	CPU/Bus	Sig.	Attr. 1	Attr. 2	Attr. 3	Other	Total	Units
TelosB	8/8MHz	18	14	24	29	8	92	ms
Imote2	13/13	1569	1421	2642	3272	784	9688	μs
Imote2	104/104	198	180	330	408	94	1209	μs
Imote2	208/208	99	89	165	204	47	604	μs
Imote2	416/208	71	62	113	136	31	413	μs

Table 5.4: Service matching latency when comparing two FFT-real service specifications

The latency of a remote search depends on the number of neighbors, the percentage of them that provide a matching service, and the order in which they are queried. Since each query is executed independently, this study evaluates a single query. Assuming the Matchmaker is on the provider, an Imote2 is used to query a TelosB to determine whether the TelosB provides a particular service. In this case, the service being queried is `FFT` and the specification is shown in Figure 5.2. It is compiled into 64 bytes, a breakdown of which is shown in Table 5.3. Performing a remote search requires the Imote2 to send the `FFT` service specification to the TelosB. Due to various bookkeeping variables used in service provisioning, the size of the query message is 72 bytes, and the reply message is 16 bytes. The time between sending the query to receiving a reply is measured by toggling a general I/O pin before and after the query, and measuring the time using an oscilloscope. Averaged over 100 trials, the latency and 90% confidence interval of determining whether a remote node has a match is $245.6 \pm 1ms$. This latency is acceptable to many WSN applications, especially since it is done only done during the service discovery process. That is, the cost of service discovery can be amortized over multiple invocations of the same service after it is bound to the task.

The latency of service binding depends on many factors. They include how many services must be considered before a match is found, the size and structure of the

specifications that are compared, where the services are located, the reliability of the wireless network when the service discovery process executes, and even the speed of the devices. Since many of these factors are unpredictable, this section analyzes the latency of determining whether two specifications match on the TelosB and Imote2 platforms.

To evaluate the efficiency of service matching, the Matchmaker is used to compare two copies of `FFT`, shown in Figure 5.2. This incurs the worst-case latency since every property within the specification must be analyzed and compared. Each experiment is repeated twenty times on both TelosB and Imote2 platforms running at all possible CPU speeds. The average latencies are calculated and the results are shown in Table 5.4.[4] The total latency consists of the latencies of comparing the service specification's signature plus each of its attributes. The column labeled "other" is the overhead incurred by the Matchmaker between comparing service specification properties. The results indicate that the TelosB takes about $92ms$ to perform a match, while the Imote2 is at least ten times faster depending on the speed setting of the CPU. As expected, the latency of comparing two specifications depends on the speed and architecture of the processor, and is mostly inversely proportional to the CPU speed, reflecting the CPU-bound nature of the comparison. The only exception is the transition from a CPU speed of 208MHz to 416MHz on the Imote2, in which the processor's data bus is the bottleneck. In all cases, the latencies are small compared to the execution times of certain VM instructions. Note that while service matchmaking does introduce overhead, it is done infrequently relative to service invocation.

To determine how the specification's size affects matching latency, `FFT` is compared to versions of itself with one, two, and all three of its attributes removed. The matching latencies is plotted against their sizes and the results are shown in Figure 5.14. For brevity, only the Imote2 running at 13MHz is shown — the latencies when the Imote2 is running at higher CPU frequencies are significantly lower and appear near zero in the figure. The results indicate that the latency is roughly proportional to its size. It is not exactly proportional because of the additional overhead incurred with the addition of each attribute, as indicated by the "other" column in Table 5.4.

[4]The confidence intervals are negligible since the experiment runs locally and the measurements exhibit very low variance.

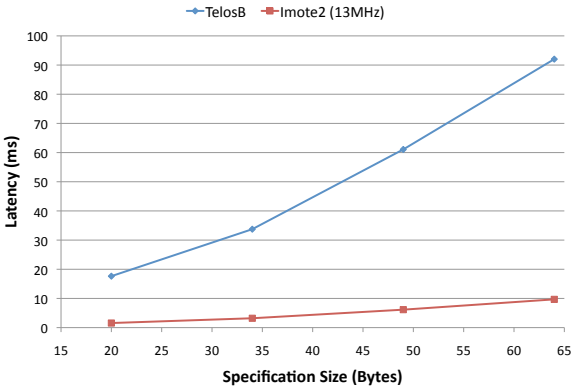


Figure 5.14: The latency of comparing a specification.

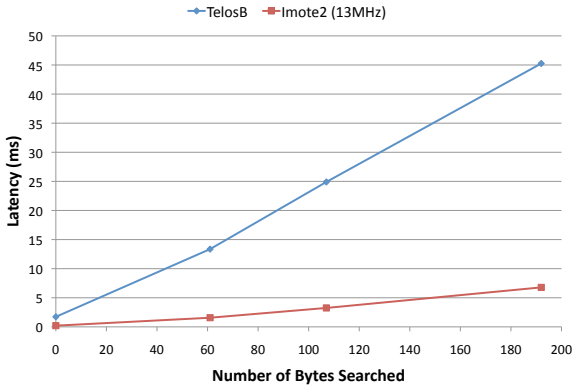


Figure 5.15: The latency of obtaining a service's binding state.

Spec.	Name	# of Properties	Size (Bytes)
1	`FFT`	3	64
2	`light-tsr`	2	46
3	`accel-3d`	5	85
4	`flash_mem`	1	34

Table 5.5: The sizes of the specifications used to evaluate service invocation

The results demonstrate the feasibility of performing service matching on resource-constrained WSN nodes.

5.6.3 Efficiency of Service Invocation

Service invocation latency can be divided into three components. The first is the time spent retrieving the binding state, the second consists of the time spent communicating over the network, and the third is the actual execution of the service. Among these, only the first two contribute to the overhead of service provisioning since the third simply executes a service that is natively-implemented. Thus, this section analyzes the latencies associated with first and second components.

The latency of obtaining a service's binding state depends on the number of specifications owned by the script, and their sizes and structure within the Specification and Binding Tables. To measure the latency of obtaining the service specification's binding state, the Specification and Binding Tables are loaded with `FFT` followed by

Spec.	1	2	3	4	Units
TelosB	2	13	25	45	ms
13MHz Imote2	206	1571	3251	6785	μs
104MHz Imote2	26	196	406	849	μs
208MHz Imote2	13	98	203	424	μs
416MHz Imote2	9	67	133	265	μs

Table 5.6: The latency of obtaining the a service's binding state

three other specifications whose properties are summarized in Table 5.5. The latency of obtaining the binding information of each specification is measured using the same technique described in Section 5.6.2, i.e., each experiment is repeated twenty times on both Imote2 and TelosB devices using every CPU frequency supported by the platform's processor. Like before, since the operations are local, the variances of the results are negligible and thus omitted. The results, shown in Table 5.6, indicate that the latency is very small with the TelosB and Imote2 devices taking up to $45ms$ and $6.8ms$, respectively. Figure 5.15 shows the linear relationship between latency versus the number of bytes searched to obtain the service's binding state. The results demonstrate the feasibility of obtaining the service binding state on a WSN node.

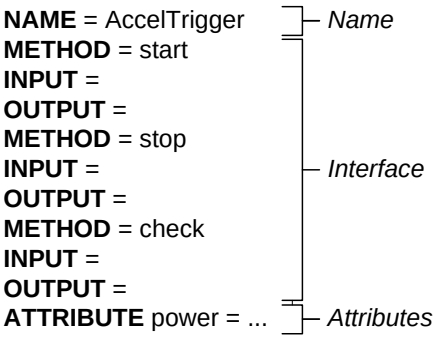
The overhead of network communication depends on the amount of data that needs to be sent and the reliability of the wireless link. To evaluate the overhead associated with network communication, Servilla's SFP is instrumented with components that measure the latency of sending a service invocation request to a device that is one hop away and receiving a response. The remote device is configured to immediately send a results message back upon receiving the request. This isolates the overhead of network communication by eliminating variability associated with the latencies of executing different services. Both the request and response contain the minimum amount of information for the SPF to operate. Specifically, the request consists of a service handle (two bytes), method handle (two bytes) and script ID (two bytes), for a total of six bytes, and the response consists of a status field indicating whether the invocation was successful (two bytes), and a script ID (two bytes), for a total of four bytes. In a normal service invocation, these messages would also include the input and output parameters of the service. In these experiments, no service is executed and no parameters are passed to isolate the overhead of network communication and eliminate service-specific overhead.

Network latency is measured by separating two devices, a service consumer (Imote2) and provider (TelosB), by approximately 0.6m in the lab. The radio power and CPU frequency are left at their default values. Averaged over 40 trials, the average and 95% confidence interval of network latency is $145 \pm 1ms$. The results exhibited very low variability due to the "clean room" state provided by the lab. A deployment in a harsh and dynamic environment will likely result in longer and more variable network latencies. Note that the network latency of service invocation is less than service discovery since less data needs to be transmitted, i.e., the service specification is not transmitted when invoking a service.

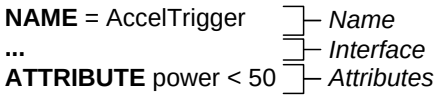
5.7 Application Case Study: Structural Health Monitoring

This section evaluates Servilla using an application case study involving structural health monitoring [33]. Structural health monitoring is a general class of WSN applications that enable continuous and real-time evaluation of a structure's integrity, reducing manual inspection costs while increasing safety. In this case study, the objective of the application is to localize damage in a structure, e.g., a bridge, based on accelerometer readings obtained from a WSN. Previously, WSNs have been used to successfully localize damage in experimental structures using a homogeneous network of Imote2 devices [66]. The implementation used an algorithm called Damage Localization Assurance Criterion (DLAC), which was written natively using NesC specifically for the Imote2 platform, meaning it is not easily ported to other platforms and does not exploit the capabilities of other types of devices. This section investigates how the application can be implemented using Servilla in a manner that improves upon the original by making it platform-independent and increasing its energy efficiency by exploiting network heterogeneity.

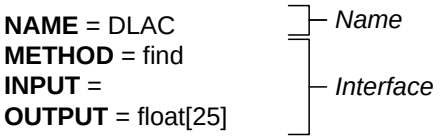
The heterogeneous WSN used in this study consists of TelosB and Imote2 devices. To enable sensing, an EasySen SBT80 sensor board is attached to each TelosB device, while an ITS400 sensor board is attached to each Imote2 device. Unfortunately, due to insufficient memory on the TelosB, the DLAC algorithm can only run on the Imote2. However, the TelosB can still be of use by monitoring whether damage is probable



(a) The specification of service `AccelTrigger` provided by Imote2 and TelosB devices. The power attribute specifies the amount of power the service consumes. It is 242mW on the Imote2, and 103mW on the TelosB.



(b) The specification of a low-power version of service `AccelTrigger`, which is provided by the application task. Its interface is omitted since it is the same as the one in Figure 5.16(a). A high-power version has attribute power $\geq$ 50 mW.



(c) The specification of service `DLAC` provided by Imote2 devices.

Figure 5.16: The services used by the damage localization application

based on the ambient vibration readings, thus enabling the Imote2 devices to remain asleep so long as there is low probability of damage. Ideally, the Imote2 devices should only be activated to perform the DLAC algorithm when the TelosB devices detect that the ambient vibration levels exceed a threshold above which damage is likely to occur. The dual-level nature of this configuration is common to other applications like surveillance [71], and is essential for conserving energy and increasing network lifetime. This section examines Servilla's ability to facilitate this heterogeneous configuration.

The Servilla implementation relies on two services: `AccelTrigger` and `DLAC`. Ambient vibrations are continuously monitored by `AccelTrigger`. When the vibrations exceed a certain preset threshold, it sets a flag indicating the high probability of damage due to the large vibrations. The specification of `AccelTrigger` is shown in Figure 5.16(a). It has three methods: `start`, `stop`, and `check`. Methods `start` and `stop` control when the service monitors the local accelerometer. Initially, and when the service is stopped, it does not monitor the local accelerometer to save energy. Only after `start`

is called does the service access the accelerometer. The status of the flag is obtained by invoking `check`. This method returns 1 if any of the ambient vibration readings obtained since the service was started have exceeded the threshold, otherwise 0 is returned. If the `check` method of this service is invoked in an event-based manner, the results are considered interesting if a 1 is returned. Both the Imote2 and TelosB devices provide `AccelTrigger`. They differ in their power attribute, since the Imote2 consumes more power than the TelosB (242mW vs. 103mW).

The specification of service `DLAC` is shown in Figure 5.16(c). It contains a single method, `find`, that takes no parameters and returns an array of floating-point numbers that are used to localize damage to the bridge [66].

Two versions of the application's task are shown in Figures 5.17 and 5.18. One version, shown in Figure 5.17, makes use of on-demand service invocations while the second version, shown in Figure 5.18, makes use of event-based service invocations. Note that the task that uses event-based invocations can also be used with periodic invocations by modifying lines 11 and 16 to call **invokePeriodic** rather than **invokeEvent**.

The first three lines of both versions of the task specify the names of the files containing the specifications of the required services. The content of `AccelTriggerLP` is shown in Figure 5.16(b), and the content of `DLAC` is shown in Figure 5.16(c). Notice that `AccelTriggerLP` matches the TelosB version of the `AccelTrigger` service shown in Figure 5.16(a) because its power attribute is less than 150mW. `AccelTriggerHP` contains the same specification as `AccelTriggerLP` except its power attribute is ≥ 150 mW, which matches the service provided by the Imote2. Note that while these service specifications match the power characteristics of platforms specific to this evaluation, they are still platform-independent in the sense that other platforms may later be introduced that provide the same service, but with different power consumption properties. Despite these differences in the power consumption property, the application can use the services provided by these new devices without modification.

The application attempts to reduce energy consumption by preferentially binding to an `Acceltrigger` service that consumes less power. It does this by first attempting to bind using the specification within `AccelTriggerLP` on line 8 of both versions of the task, before using the specification within `AccelTriggerHP` on line 13 of both versions of the task. Once an `AccelTrigger` service is bound, the two tasks differ

in how they invoke the service. The task shown in Figure 5.17 performs on-demand service invocation, meaning it must periodically query the service to determine if the acceleration readings are above a certain threshold (lines 21-37). If it is, as indicated by a return result of 1, `DLAC` is invoked and the results are sent to the base station (lines 39-43). The task shown in Figure 5.18 performs event-based service invocation, meaning it does not need to periodically query the service. The key lines of code are 11 and 16 of Figure 5.18. The lines specify that the "check" method of the service should be invoked periodically once every second, as indicated by the 1024 parameter, and the name of the event-callback function that should be called when "check" returns an interesting result. In this case, the event-callback functions for the low and high power services are "lpEvent" and "hpEvent", respectively. Both of these callback functions verify that the return value of the service invocation is one (lines 22 and 29). If it is, the `AccelTrigger` service is stopped (lines 23 and 30), and the DLAC algorithm is invoked (lines 35-39). Note that another reason why the event call-back functions check the flag is to enable the task to be easily modified to use periodic invocations, as described above.

To evaluate the benefit of exploiting network heterogeneity on Servilla, the tasks shown in Figures 5.17 and 5.18 are injected into two WSNs: a homogeneous network consisting of only Imote2 devices, and a heterogeneous network consisting of both Imote2 and TelosB devices. In addition, a modified version of the task shown in Figure 5.18 that uses periodic service invocations is also used to enable comparisons among all three forms of service invocation. Since the application is written using Servilla, it is able to run on both types of networks without modification. In all cases, `DLAC` is executed by the Imote2, meaning the power consumption of performing damage localization is constant. However, the power consumption of `AccelTrigger` varies because Servilla's service provisioning framework enables an application to exploit more energy-efficient services when possible in a platform-independent and declarative fashion. Specifically, if TelosB devices are present, the service will be executed on a TelosB device since its `AccelTrigger` service consumes less power, otherwise it will be executed on the Imote2. We compare the power consumption of invoking `AccelTrigger` in different network configurations and using different forms of service invocations.

Action/State	TelosB	Imote2	Units
Idle Power (Sensor Off)	0.45	109.7	mW
Idle Power (Sensor On)	–	204.83	mW
Sensing	102.9	241.83	mW
Sensing Latency	18.49	2.61	ms
Message Tx Power	51.82	184	mW
Message Tx Latency	725	506	ms
Message Rx Power	57.52	192.91	mW
Message Rx Latency	6.17	14.07	ms

Table 5.7: Power and latency attributes of TelosB and Imote2 platforms when radio is operating at 1% duty cycle.

Since invoking `AccelTrigger` on the TelosB requires a remote invocation, the power draw depends on the invocation period in addition to the sensing frequency. The invocation period is the time between each invocation of the "check" method, while the sensing frequency is the rate at which the accelerometer is accessed. If the service is invoked too often, a larger percentage of energy will be spent on wireless communication. Likewise, if the sensor is accessed very infrequently, the benefits of the TelosB is diminished since the devices will remain asleep a larger percentage of the time. To determine how much energy savings are possible, the power draws and latencies of performing various operations and operating in certain states are obtained. This is done using a Tektronix TDS 2004B digital oscilloscope simultaneously measuring the voltage across a high-accuracy resistor placed in parallel with the device, and the voltage drop across the device itself. The measurements obtained and used in the remainder of this section are shown in Table 5.7. As expected, the TelosB draws far less power than the Imote2. In some cases, like idling, it draws several orders of magnitude less power. Note that the Imote2 has two idle powers, one with the sensor board on and another with the sensor board off. This is due to limitations of the ITS400 driver that prevents the sensor from being turned off between readings. The consequence is a magnification of the benefits of periodic and event-based invocations, which require fewer message transmissions. All of the measurements when the radio was being operated at a 1% duty cycle using an asynchronous duty cycling mechanism, which is included in TinyOS. While this enables lower idling power, due to the radio being turned off 99% of the time, it results in long latencies when sending messages.

In this case study, the invocation period is varied between 1 to 120 seconds and sensing frequency is varied between 1 and 14Hz. The selection of the invocation period range is done mostly arbitrarily — the only restriction is that it not be less than the round-trip communication time, which in this case is $725ms + 6.17ms = 731.17ms$. The selection of 14Hz as the maximum sensing frequency is to prevent overloading the TelosB device, which operates slower than the Imote2. Consider the fastest service invocation period of 1s. In this case, the percent utilization of the TelosB dedicated to handling the invocation attempts is given by Equation 5.1.

$$\frac{\texttt{Message Tx latency} + \texttt{Message Rx latency}}{\texttt{invocation period}} = \frac{0.00617 + 0.725}{1} = 0.7312 \quad (5.1)$$

Equation 5.1 indicates that handling invocation requests utilizes a maximum of 73.12% of the processor. Thus, sensing can consume up to $100\% - 73.12\% = 26.88\%$ of the processor. The processor utilization of sensing is given by Equation 5.2.

$$\frac{\texttt{Sensing Latency}}{\texttt{Sensing Period}} = 0.01849 \cdot \texttt{Sensing Frequency} \quad (5.2)$$

Setting Equation 5.2 to have a maximum value of 26.88% and solving for the sensing frequency derives a maximum sensing frequency of 14.55Hz, thus the selection of 14Hz as the maximum sensing frequency used in this case study.

The percent reduction in power utilization of using a heterogeneous versus a homogeneous network is calculated for the three service invocation techniques and the range of invocation periods and sensing frequencies described above. The homogeneous network configuration consists of the Imote2 invoking the service locally, while the heterogeneous network consists of the Imote2 invoking the service on the TelosB. All calculations assume that no vibrations exceeding the preset threshold have been detected, meaning they reflect the power savings when the system is running in long-term steady state. This is reasonable since exceptional events like damage-inducing vibrations are expected to be rare. The results are shown in Figure 5.19. They show that regardless of the type of invocation used, exploiting network heterogeneity using Servilla results in a reduction in power utilization under most circumstances. The

only exception is when on-demand service invocations are issued at the fastest period (1 second) and sensing frequency (14Hz), in which case a heterogeneous network draws 6.8mW more power, as shown by the negative portion in Figure 5.19(a). In all other configurations, using a heterogeneous network *always* results in significant reductions in power consumption.

For example, on-demand and periodic service invocations both converge towards the same reduction in power as the invocation period increases, with a power reduction ranging from 33% to 45% depending on the sensing frequency. This is shown in Figures 5.19(a) and 5.19(b). They converge because the only difference between them is the need for the an invocation message to be sent each time the service is invoked. This difference becomes increasingly negligible as the invocation period increases. That is, there is a limit to the amount of energy that can be saved as the service invocation period increases since it approaches the difference between the energy consumed by the Imote2 versus the TelosB idling and sensing at the prescribed rate. Note that the percent reduction in power consumption is less when the sensor is accessed more rapidly. This is because accessing the sensor more often results in the node being able to idle less, and the difference in the idle power draws of the TelosB and Imote2 is greater than the difference in the sensing power draws.

Event-based invocations result in an even more significant reduction in power consumption at 87% to 98.9%, as shown in Figure 5.19(c). Event-based invocations result in large reductions in power draw since they do not require any messages to be sent so long as no event of interest has occurred. Thus, the percent reduction in power reflects the difference between the power draw of the Imote2 versus the TelosB accessing the sensor. Note that the reduction in power consumption remains constant regardless of the invocation period because invocations do not involve sending any messages in either direction (this is assuming the system is operating in steady state in which no interesting event has occurred).

To understand the relative differences between the various forms of remote invocation in a heterogeneous network, the reduction in power consumption when selecting different forms of remote invocation are shown in Figure 5.20. Figure 5.20(a) and 5.20(b) show the percent reduction in power utilization when using event-based invocations

relative to on-demand and periodic invocations. In both cases, event-based invocations result in significant energy savings since messages do not need to be transmitted so long as no interesting event has been detected. This is reflected by the fact that as the invocation period decreases, the percent savings increase due to the additional messages being sent in the on-demand and periodic forms of invocation.

The similarities between Figures 5.19(a) and 5.19(b) and Figures 5.20(a) and 5.20(b) may lead one to conclude that there is little difference between on-demand and perodic invocations in a heterogeneous WSN. To determine the difference between these two forms of invocation, Figure 5.20(c) shows the percent reduction in power draw when performing periodic invocations relative to on-demand invocations. It shows that there always exists savings, and that the savings increase dramatically as the invocation period decreases. This makes sense since decreasing the invocation period increases the number of invocations per unit time. Since periodic invocations do not need an invoke message to be sent each time the service is invoked, a greater reduction in energy savings is obtained by using periodic invocations.

This case study demonstrates how Servilla enables platform-independent applications that operate over a heterogeneous WSN, and how it facilitates in-network collaboration between different types of devices to attain higher energy efficiency. Moreover, it demonstrates that Servilla enables an application to bind to a more energy-efficient service through service specification, and that the different forms of service invocation significantly impact the attainable energy savings. Note that this case study demonstrated one of many implementations of the application. An alternative implementation is to divide the application into two tasks. One would run on a low-power node and invoke `AccelTrigger`, while another would run on a high-power node and execute DLAC when the low-power task notifies it of potential damage. This implementation requires that the low-power node be able to run a VM and SPF, which would exclude the TelosB. Thus, the current implementation was selected since it enables weaker nodes like the TelosB to contribute to the application. The analysis demonstrated that the decision to include the weaker TelosB node results in significant reductions in power consumption in most scenarios.

5.8 Chapter Summary

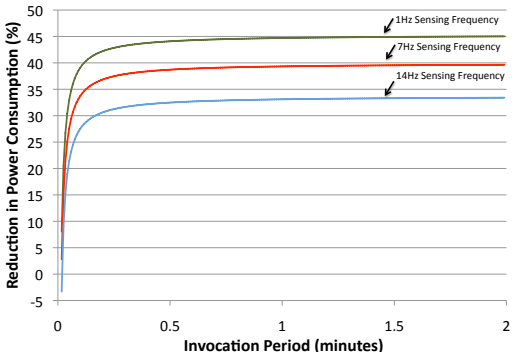
The increasing difficulty of developing applications for heterogeneous and dynamic WSNs demands a new coordination model. Servilla provides this by introducing a novel service provisioning framework that enables applications to be platform-independent while still able to access platform-specific capabilities. A salient feature of Servilla lies in its capability to support coordination and collaboration among heterogeneous devices *inside* a WSN. A specialized service description language is introduced that enables flexible matching between applications and services, which may reside on different devices. Servilla provides a modular middleware architecture to enable resource-poor devices to participate by contributing services, facilitating in-network collaboration among a wide range of devices. The efficiency of Servilla's implementation is established via microbenchmarks on two representative classes of hardware platforms. The effectiveness of Servilla's programming model is demonstrated by a structural health monitoring application case study.

```
1.  uses AccelTiggerHP;
2.  uses AccelTiggerLP;
3.  uses DLAC;
4.
5.  void main() {
6.      bind(DLAC, 0); // bind DLAC service
7.      if(!isBound(DLAC)) exit(); // failed to bind DLAC
8.      bind(AccelTriggerLP, 1); // bind low-power AccelTrigger service
9.      if(isBound(AccelTriggerLP)) {
10.         invoke(AccelTriggerLP, "start");
11.         waitForTrigger(1);
12.     } else {
13.         bind(AccelTriggerHP);
14.         if(isBound(AccelTriggerHP)) {
15.             invoke(AccelTriggerHP, "start");
16.             waitForTrigger(0);
17.         }
18.     }
19. }
20.
21. void waitForTrigger(int useLowPower) {
22.     int vibration = 0;
23.     while(vibration == 0) {
24.         if (useLowPower)
25.             vibration = invoke(AccelTriggerLP, "check");
26.         else
27.             vibration = invoke(AccelTriggerHP, "check");
28.         if (vibration == 1) {
29.             if (useLowPower)
30.                 invoke(AccelTriggerLP, "stop");
31.             else
32.                 invoke(AccelTriggerHP, "stop");
33.             doDLAC();
34.         }
35.         sleep(1024*60*5); // sleep for 5 minutes
36.     }
37. }
38.
39. void doDLAC() {
40.     float[25] dlac_data;
41.     dlac_data = invoke(DLAC, "find");
42.     send(dlac_data); // send DLAC data to base station
43. }
```

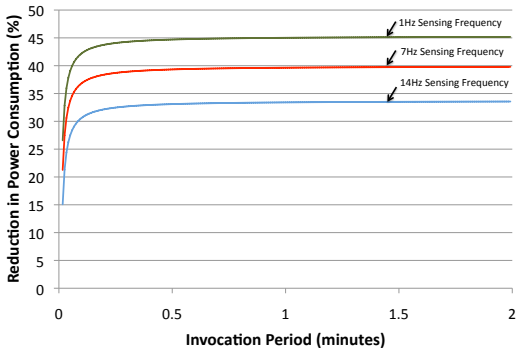
Figure 5.17: The damage localization application task using on-demand invocations

```
1.  uses AccelTiggerHP;
2.  uses AccelTiggerLP;
3.  uses DLAC;
4.
5.  void main() {
6.      bind(DLAC, 0); // bind DLAC service
7.      if(!isBound(DLAC)) exit(); // failed to bind DLAC
8.      bind(AccelTriggerLP, 1); // bind low-power service
9.      if(isBound(AccelTriggerLP)) {
10.         invoke(AccelTriggerLP, "start");
11.         invokeEvent(AccelTriggerLP, "check", 1024, lpEvent);
12.     } else {
13.         bind(AccelTriggerHP,1);
14.         if(isBound(AccelTriggerHP)) {
15.             invoke(AccelTriggerHP, "start");
16.             invokeEvent(AccelTriggerHP, "check", 1024, hpEvent);
17.         }
18.     }
19. }
20.
21. void lpEvent(int flag) {
22.     if(flag == 1) {
23.         invoke(AccelTriggerLP, "stop");
24.         doDLAC();
25.     }
26. }
27.
28. void hpEvent(int flag) {
29.     if(flag == 1) {
30.         invoke(AccelTriggerHP, "stop");
31.         doDLAC();
32.     }
33. }
34.
35. void doDLAC() {
36.     float[25] dlac_data;
37.     dlac_data = invoke(DLAC, "find");
38.     send(dlac_data); // send DLAC data to base station
39. }
```

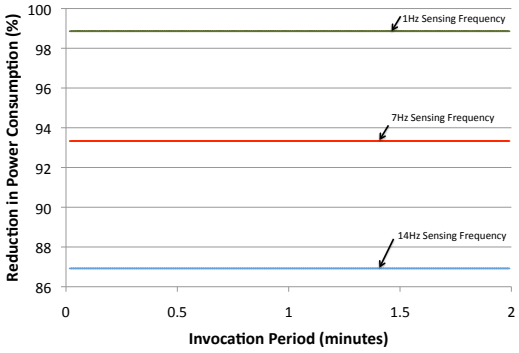
Figure 5.18: The damage localization application task using event-based invocations



(a) On-Demand Service Invocation.

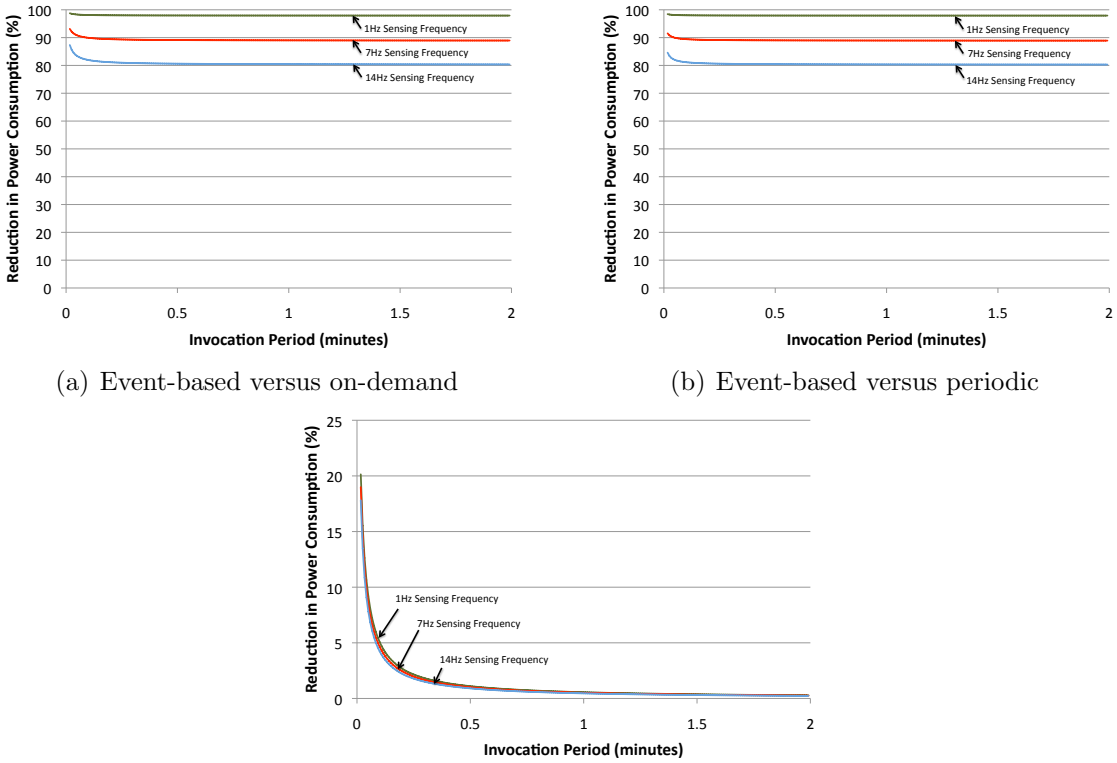


(b) Periodic Service Invocation.



(c) Event-Based Service Invocation

Figure 5.19: Percent power savings of heterogeneous vs. homogeneous WSNs.



(a) Event-based versus on-demand

(b) Event-based versus periodic

(c) Periodic versus on-demand

Figure 5.20: Relative power savings of different invocation types on heterogeneous WSNs.

Chapter 6

Servilla Extension: Adaptive Service Provisioning

The previous chapter presented Servilla, a middleware that simplifies application implementation by exploiting the decoupling provided by service provisioning to elegantly handle device heterogeneity in WSNs. This chapter further explores how this decoupling can enhance applications in terms of their ability to adapt to network dynamics and promote energy conservation. The key observation is that novel adaptive service binding strategies can cope with network dynamics and promote energy conservation. To achieve this, Servilla is extended to include policies and algorithms that automatically adapt application behavior when opportunities for energy savings arise, and switch providers in response to network topology changes. The former is accomplished by providing limited information about the energy consumption associated with using various services, by systematically exploiting opportunities for sharing service invocations, and by exploiting the broadcast nature of wireless communication in WSNs. The latter is accomplished by monitoring the performance of the current and alternative service providers. The policies and algorithms have been implemented and evaluated on two disparate WSN platforms, the TelosB and Imote2. Empirical results show that adaptive service provisioning can enable energy-aware service binding decisions that result in increased energy efficiency and significantly increase service availability, while imposing minimal additional burden on the application, service, and device developers. The system's efficacy is demonstrated through two application case studies: medical patient monitoring and structural health monitoring.

6.1 Motivation

Service-Oriented Computing (SOC) [137] has traditionally been used to enable independently created software to work together. In the context of WSNs, it was initially used as a way to rapidly integrate data obtained by WSN nodes with Internet-based applications [147, 9, 89]. More recently, SOC was used to facilitate the development of applications that run *within* WSNs that consist of a heterogeneous mix of devices, as described in Chapter 5 [157, 93]. While these usages of SOC are useful, the systems that implement them relied on traditional service binding schemes that were originally designed for use on the Internet, though with significantly simplified implementations. They are not tailored to the unique properties of WSNs like limited energy and unreliable network connectivity. This chapter proposes novel service binding schemes that enhance energy efficiency and service availability in an autonomous and application-transparent manner.

Two novel service binding schemes are proposed. First, novel service selection strategies are developed that enhance the energy efficiency of a WSN. This is important because when multiple providers are available in a heterogeneous network, each may be configured differently resulting in the varying energy efficiencies. Thus, the selection of a provider affects the energy footprint of an application. To make the SOA energy-aware, a limited amount of information regarding a provider's energy efficiency is included in the provider's response to a service discovery request, allowing the consumer to determine which provider will result in the highest energy efficiency. Furthermore, opportunities for sharing service executions are automatically identified and exploited to increase energy efficiency. This is particularly useful when combined with the broadcast nature of wireless communication, which enables the results of a single execution to be simultaneously delivered to multiple consumers.

Second, adaptive service binding strategies are developed to automatically adjust the bindings between service providers and consumers in response to changes in the network topology. This is important because WSNs exhibit high levels of dynamics due to node mobility, exposure to a harsh and dynamic environment, and the use of low-power radios susceptible to fluctuations in link quality [64]. A key advantage of an adaptive service binding scheme is that it enables *application-transparent* handling of

network topology changes in a SOC framework, and thus greatly simplifies application development despite network dynamics.

Significant contributions of this work also lie in the implementation of the adaptive service binding strategies and comprehensive empirical evaluation through both microbenchmarks and two application case studies. Specifically, the adaptive service binding strategies are implemented in an enhanced version of Servilla that runs on TinyOS 2.1. It was evaluated on two disparate hardware platforms, the Imote2 [37] and TelosB [144], which differ significantly in terms of energy efficiency motivating the need for energy-aware adaptation mechanisms.

To evaluate the efficacy, feasibility, and usability of our adaptive SOA, a detailed analysis of how the adaptation mechanisms are configured for the Imote2 and TelosB platforms is performed. They indicate that the adaptation mechanisms do *not* impose undue additional burden on the device, service, and application developers. In addition, two real-world application case studies involving medical patient monitoring and structural health monitoring are implemented. The medical patient monitoring application involves a user moving through a region covered by a 74-node WSN testbed spread across two buildings at Washington University in St. Louis [175] periodically invoking services provided by nodes in the testbed. Adaptive service provisioning achieved 100% service invocation success rate despite frequent topology changes caused by user mobility. The structural health monitoring application involves a WSN dedicated to detecting and localizing damage in a structure. It demonstrates the ability of adaptive service provisioning to enhance energy efficiency through energy-aware service selection and sharing.

6.2 Related Work

SOC has been used in WSNs for various purposes [119]. One original use is to integrate WSNs with Internet applications [9, 147, 146, 102, 10, 160]. To do this, the WSN is hidden behind services that provide sensor data. Using SOC, Internet applications can bind to these services and access information generated by the WSN. Application development is simplified since developers are already familiar with SOC

programming. Systems that provide this differ in the degree to which SOC is integrated, and the operations that are performed. For example, PhyNet™ [9] and TinySOA [10] implement a single provider as a translation layer on the WSN gateway, which interfaces between the WSN and an IP network. Prinsloo et. al. [146] presents a system that provides the Open Service Gateway Initiative (OSGi) [135] SOA on the gateway. Li et. al. [102] and Sommer et. al. [160] present advanced query re-writing and code generation functionalities on the gateway for increasing the energy efficiency of service invocations and service reuse. In these systems, the protocols that transfer the sensor data from each WSN node to the gateway are proprietary but hidden from the application developer. Tiny Web Services (TWS) [147] implements service providers on each WSN node, enabling new types of services to be added without modifying the gateway. While these systems represent major steps toward the integration of WSNs with the Internet, they provide traditional service binding schemes that are not specifically designed to enhance energy efficiency and service availability, which are common concerns in WSNs.

In addition to integrating WSNs with traditional networks, SOC has also been used for enabling adaptation to network heterogeneity. This is exemplified by Servilla, the SOA that was described in Chapter 5. Its key idea was to present platform-specific functionalities as services that are dynamically bound to platform-independent applications. Servilla differs from the system presented in this paper in that it does *not* provide adaptive service provisioning. Service binding and unbinding is done explicitly by the application and energy efficiency is not automatically considered when selecting a provider — the application had to include it as a required attribute in a service specification, and manually select the most energy-efficient provider. This is problematic because it assumes that the consumer has knowledge about the energy consumption of the potential providers at the time the application is written. Other systems that use SOC to adapt to network heterogeneity include eSOA [157] and OASiS [93]. They differ from the system described in this chapter by performing service matching and binding *off-line* on the base station.

In-network reprogramming [91, 101, 138, 115] enables adaptation via code updates. By replacing the code in the WSN, almost any form of application behavior can be added, resulting in maximum adaptation flexibility. However, they differ from the system presented in this paper in that the adaptation decisions are made by the user

at a centralized gateway and require disseminating code from the gateway onto the WSN nodes, which is an energy-intensive process. In contrast, the system presented in this paper enables each consumer to automatically make local adaptation decisions in an energy-efficient manner.

Macro-programming [74, 12, 63, 177] is another mechanism for adaptation in WSNs. It enables application developers to treat the entire WSN as if it were a single device by automatically decomposing the application written by the developer into micro-programs that are distributed among the WSN nodes. Adaptation capabilities are achieved via the decomposition process, i.e., it adjusts the decomposition based on the WSN topology in a manner that is transparent to the user. A key difference between macro-programming systems and the system presented in this chapter is the fact that the adaptation is done at compile-time before the micro-programs are deployed onto individual WSN nodes. The adaptive SOA presented in this chapter performs on-line adaptation within the WSN.

Energy efficiency is another key focus of this paper. It is so important in the context of WSNs that nearly every aspect of the WSN software stack, from the MAC layer via duty cycling to the application layer via data aggregation and adaptive sensor sampling rates, contains mechanisms for increasing energy efficiency [4]. Numerous WSN systems focus on energy efficiency. For example, Santini et. al. [156] presents an adaptive algorithm for predicting sensor data readings, enabling energy to be conserved by decreasing the amount of sensor data that needs to be transmitted. It differs from the system presented in this paper by focusing on optimizing a specific type of data (sensor readings), whereas an adaptive SOA optimizes operations performed by services in general. Given the necessity to consider energy consumption in all aspects of WSNs, making the SOA energy-aware is essential. Unlike previous systems that increase energy efficiency, the system presented in this paper uniquely focuses on how energy can be saved through careful service selection and opportunistically merging service executions.

6.3 Problem Definition

The two problems addressed in this chapter are how SOC can be used to enable applications to 1) conserve energy, and 2) transparently adapt to changing network topologies. This section explains the system model and presents the design goals.

6.3.1 System Model

The system consists of a WSN in which there are consumers and providers. Consumers are controlled by applications that require and invoke services. Providers provide services that are dynamically discovered, bound to, and invoked by consumers. The consumers and providers communicate locally when they share the same node, or over a wireless link when they are located on different nodes. The limited wireless range results in a consumer only being able to communicate with a subset of all matching providers in the network. Since wireless links change over time, this subset of matching providers is dynamic. Currently only single-hop service invocations are supported.

Once a consumer binds to a provider, it can invoke the provider's service by sending an "invoke message" that initiates the invocation. Upon receiving this message, the provider executes the service, and replies with a "results message" that contains the invocation results. Depending on the type of invocation being performed, the provider may repeat this process a certain number of times.

Switching providers is assumed to involve no state transfer from the old provider to the new. This simplifies the SOA by eliminating any intrinsic overhead associated with switching providers. That is, assuming the set of matching providers is known, the SOA can arbitrarily switch between any provider within the set without incurring additional energy relative to using the same provider continuously. Furthermore, this assumption enables the SOA to react to network disconnection, which is important for ensuring energy efficiency. Many services like sensing and data routing can be offered in a manner that meets this assumption, though some services like data storage cannot. In the future, this assumption can be removed by including the overhead of

state transfer in the energy consumption computations, and implementing a mechanism that determines when a provider is about to disconnect so the data can be transferred before actual disconnection occurs.

WSNs are different from traditional networks in that they are energy-limited and rate-based. They often remain idle until a particular event like the detection of a phenomenon occurs. To account for these differences, SOC in WSNs have three forms of service invocations: *on-demand*, *periodic*, and *event-based*. On-demand is what is traditionally provided by most SOAs in which an invocation is similar to a remote procedure call. That is, the consumer initiates a service invocation by sending the provider a message, and waits for the provider to respond with results. Unfortunately, the two-way message exchange is energy inefficient if the service needs to be invoked many times. To account for this, periodic and event-based invocations involve the provider automatically invoking the service periodically. They differ in that periodic invocations send every result back whereas event-based invocations only send interesting results, as defined by the provider, back to the consumer. Both forms of invocations are more energy efficient since they do not require the consumer to send the provider a message each time the service is invoked.

While most nodes operate on batteries and are energy-constrained, some nodes are not. For example, in a medical patient monitoring application, a network of nodes that relay data from the patient to the nurse's central monitoring station can be embedded in the walls and ceilings of the hospital, enabling them to be powered by the building's electrical grid [34]. Since not all nodes are energy-constrained, the SOA must consider this fact when accounting for energy costs. Clearly, nodes that are not energy constrained should not be included in the energy cost calculations.

6.3.2 Design Goals

The following are the primary objectives of the adaptive SOA presented in this chapter.

- Reduce energy consumption through energy-aware service selection and sharing. The selection of a particular provider affects the amount of energy consumed

due to device heterogeneity and differences in wireless link qualities between the consumer and provider. Achieving this objective involves developing an algorithm that determines *which* provider to select when switching providers.

- Enhance service availability through application-transparent service rebinding. This is necessary due to the transient connectivity between the consumers and providers. Achieving it requires developing an algorithm that determines *when* to switch providers.

The objective of the adaptation mechanism is to hide provider disconnection from the application. Thus, the adaptation mechanism should prevent the application from being exposed to service invocation failure when other potential providers exist within its neighborhood. Achieving this requires solving different problems depending on the invocation type. Specifically, a successful adaptation mechanism must ensure that the results of an on-demand invocation are *always* returned successfully. For periodic invocations, the number of invocation results received must be the number expected. For event-based invocations, the service must be continuously invoked in a periodic manner despite changes in the actual provider providing the service.

In addition to network topology changes, the adaptive mechanism should also conserve energy. In this chapter, this is done by reducing an application's "energy footprint," which is the total energy an application consumes invoking services. This includes the energy spent on wireless communication and service execution on all nodes in the network (including the hosts of both consumers and providers) that are energy-constrained.

So far, the only problems identified are those related to the adaptation algorithms themselves. A few additional problems must be solved in terms of the SOA's overall usability and practicality. The first is how to ensure the system is responsive in terms of adapting to network topology changes. Second, the problem of additional overhead for achieving adaptation must be addressed. Specifically, it must not outweigh the energy efficiency gained through adaptation. Finally, the problem of additional burden imposed on the application, device, and service developers must be addressed. Ideally, their software components can be integrated with the adaptive SOA without any changes.

6.4 Adaptation Mechanisms

This section presents the adaptation mechanism developed and integrated into a SOA for WSNs. Before presenting the details, an overview of the basic service selection and binding process is given. Service selection is the process of selecting one provider from among the set of all known providers that provide the desired service. It involves the consumer analyzing the properties of each provider and selecting the one that it believes best meets its requirements. Upon selecting a particular provider, the consumer binds to it by noting the provider's address. This address is used to communicate with the provider when the consumer invokes the service. Note that the address of the provider is hidden from the application by the SOC middleware. The application is presented with a simple interface enabling it to invoke the service.

The remainder of this section is divided into three parts: 1) selecting the most energy-efficient provider, 2) optimizing energy efficiency via shared service invocations, and 3) increasing service availability by adapting to network topology changes.

6.4.1 Energy-Aware Provider Selection

This section describes *which* provider to select. Provider selection must be energy-aware since it impacts energy consumption due to differences in hardware architectures and wireless link qualities. For example, the Imote2 and TelosB differ widely in terms of power draw, i.e., 145mW versus 9mW. Thus, binding to an Imote2 can potentially result in an order of magnitude greater energy consumption relative to a TelosB.

Fundamentally, deciding which provider to select is simple: choose the one that results in the smallest energy footprint. The problem is how the energy footprint of a particular binding can be determined. Doing this requires analyzing the various steps of invocation, which depends on the type of invocation being performed and whether the provider is local or remote.

First consider on-demand and periodic invocations. On-demand invocations are a special case of periodic invocations in which the number of periods is one. Thus,

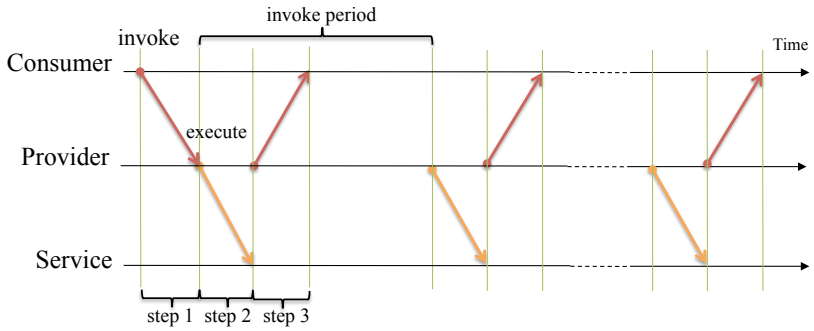


Figure 6.1: The actions performed during periodic invocations.

both forms of invocation share the same three basic steps: 1) initiation, 2) execution, and 3) results delivery. Figure 6.1 contains a visualization of these steps. Initiation involves the consumer telling the provider that it wants to invoke the service. If the provider is local, this consumes negligible energy since it essentially amounts to a method call. However, if the service is remote, this involves the consumer sending an invoke message to the provider. Execution involves actually running the service. This includes all energy associated with executing the service. Finally, results delivery involves the provider sending the results of the invocation to the consumer. Like the first step, the energy consumption is negligible if the provider is local, but involves one message transmission if it is remote. For periodic invocations, the last two steps are repeated as specified by the consumer.

To determine the energy footprint of a particular binding state, each step of the service invocation process must be analyzed. The variables used in the analysis are shown in Table 6.1, and the energy footprint for on-demand and periodic invocations is given by Equation 6.1. Note that the values of the variables in Table 6.1 are based on the specific hardware used, the derivation of which will be described in Section 6.5.

$$\begin{aligned} E_{periodic} = {} & E_{tx,c} + E_{rx,p} + (\texttt{InvokeCount}) \cdot (P_{idle,c} \cdot T_{invoke} \\ & + T_{invoke} \cdot P_{invoke} + E_{rx,c} + E_{tx,p}) + (\texttt{InvokeCount} - 1) \\ & \cdot (P_{idle,c} \cdot (\texttt{InvokePeriod} - T_{invoke} - T_{rx,c}) \\ & + P_{idle,p} \cdot (\texttt{InvokePeriod} \cdot T_{invoke} - T_{tx,p})) \end{aligned} \tag{6.1}$$

Application Developer

Symbol	Meaning	Units
`InvokePeriod`	Invocation Period	ms
`InvokeCount`	Number of invocations	n/a

Service Developer

Symbol	Meaning	Units
T_{invoke}	Latency of one service execution	ms
P_{invoke}	Power during service execution	mW

Device Developer

Symbol	Meaning	Units
$T_{rx,c}$	Latency of consumer receiving a packet	ms
$T_{tx,p}$	Latency of provider sending a packet	ms
$P_{idle,c}$	Power when consumer is idle	mW
$P_{idle,p}$	Power when provider is idle	mW
$E_{tx,c}$	Energy cost of consumer sending a message	μJ
$E_{tx,p}$	Energy cost of provider sending a message	μJ
$E_{rx,c}$	Energy cost when consumer receives a message	μJ
$E_{rx,p}$	Energy cost when provider receives a message	μJ

Table 6.1: Variables for deriving the energy cost of service invocation, and who must supply them.

The first two variables of Equation 6.1 account for the energy in step one of the service invocation process, i.e., when the consumer sends a message to the provider indicating that the service should be invoked. The middle part of Equation 6.1 consists of a product capturing the energy consumed by both the consumer and provider when the service is actually invoked. This includes both the energy to execute the service and deliver the results to the consumer. Note that it is multiplied by `InvokeCount` since that is the number of times the service is invoked. Finally, the last part of Equation 6.1, accounts for the energy consumed when the consumer and provider is idling between service invocations. It is multiplied by (`InvokeCount` $- 1$) because after the last invocation is completed, the process is considered completed.

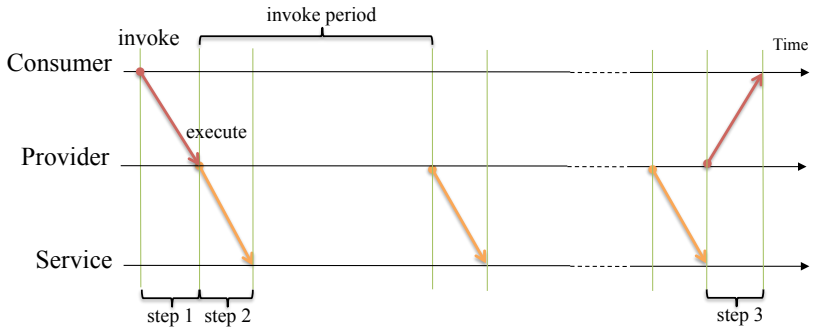


Figure 6.2: The actions performed during event-based invocations.

When the binding is local, equation 6.1 can be simplified. This is because in this case there is no energy cost associated with wireless communication. Specifically, the equation for local invocation is as follows:

$$E_{local} = \texttt{InvokeCount} \cdot T_{invoke} \cdot P_{invoke} + (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) \cdot P_{idle} \quad (6.2)$$

There is no designation of whether P_{idle} in Equation 6.2 is a consumer or provider, since the same node plays both roles in a local invocation. In addition, equation 6.2 captures the energy footprint of all forms of local invocation, including event-based invocations, since there is no network communication cost.

The energy cost of remote event-based service invocations is different from that of remote periodic invocations since uninteresting results are not sent to the consumer. To visualize the difference, the sequence of actions performed during remote event-based service invocations is shown in Figure 6.2. The first step consists of the consumer sending a message to the provider that initiates the service invocation process. Upon receiving this message, the provider periodically executes the service on behalf of the consumer, but analyzes the results and avoids sending those that are uninteresting. This process repeats until an interesting result is obtained, which is sent back to the consumer, concluding the invocation process. Note that in an actual deployment, the number of invocations that must occur before an interesting one is found may not be known. In this case, the application programmer must estimate the *likely*

number of service executions necessary before one of interest occurs. The equation for deriving the energy footprint of remote event-based service invocations is given by Equation 6.3.

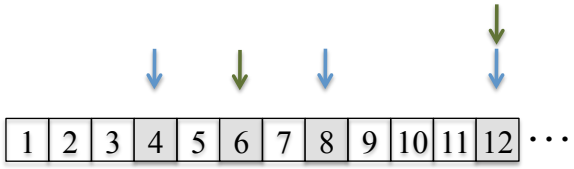
$$\begin{aligned} E_{event} = {} & E_{tx,c} + E_{rx,p} + E_{tx,p} + E_{rx,c} \\ & + \texttt{InvokeCount} \cdot (P_{idle,c} \cdot T_{invoke} + T_{invoke} \cdot P_{invoke}) \\ & + (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) \cdot (P_{idle,c} + P_{idle,p}) \end{aligned} \tag{6.3}$$

The first line of equation 6.3 captures the energy consumed during steps one and three of the service invocation process in which the initial message initiating the invocation is sent and the results are delivered back to the consumer. The second line captures the energy spent in step two where the service is executed by the provider. Finally, the third line captures the energy spent idling between service invocations.

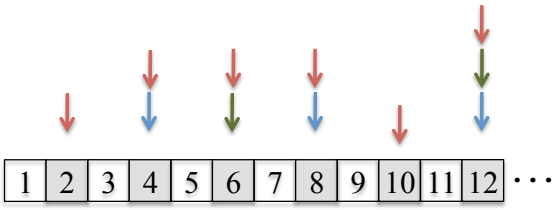
By implementing equations 6.1, 6.2, and 6.3 and integrating it into the middleware's service discovery and selection mechanism, the set of matching providers can be automatically sorted based on the amount of energy they will consume if selected. This enables the adaptation mechanism to select the provider that will result in the smallest energy footprint, which is essential in energy-constrained WSNs.

To capture situations in which a node is not energy-constrained, the energy cost of the node can simply be set to zero. Equations 6.1, 6.2, and 6.3 can still be used without modification. For example, if the provider is line-powered, $E_{tx,p}$, $E_{rx,p}$, and P_{invoke} should be set to zero. This will effectively remove non-power-constrained nodes from the energy cost calculation.

As mentioned in Section 6.3, an important requirement of the adaptive middleware is that it does not impose too much burden on the device, application, and service developers. In this case, the additional burden is the derivation of the variables shown in Table 6.1. To understand the actual amount of additional work required of each party, the variables shown in Table 6.1 are divided based on who needs to provide them. The device developer needs to specify eight variables related to the energy efficiency and latency of wireless communication and idling. This only needs



(a) $P_1 = 4$, $P_2 = 6$



(b) $P_1 = 4$, $P_2 = 6$, $P_3 = 2$

Figure 6.3: A visualization of how service utilization is calculated.

to be done once for each platform type. The service and application developers each need to specify only two additional variables. In the application developer's case, the two variables, `InvokeCount` and `InvokePeriod`, need to be specified anyway when invoking a service periodically or in an event-based manner. In other words, in most circumstances, there is *no additional burden* placed on the application developer when enabling adaptive capabilities. The feasibility of deriving these values is shown in Section 6.5, while the validity of the equations are shown in Section 6.6.

6.4.2 Efficiency Through Invocation Sharing

Periodic and event-based invocations execute a service once every period, making the timing of the next invocation predictable. This enables a novel mechanism for saving energy: *service invocation sharing*. The idea is that in certain situations multiple service execution requests can be combined into one. In addition, depending on whether reliability is needed, the results of the single service execution can be delivered to multiple consumers simultaneously via wireless broadcast. By reducing the number of times a service needs to be executed and the results delivered, energy savings is possible. This section investigates this possibility.

To understand how energy can be saved via service sharing, consider the impact a particular invocation has on a service's utilization assuming sharing is possible, as shown in Figure 6.3. Time is discretized into an array of boxes in which each box may or may not execute the service. Thus, when a consumer invokes a service periodically or in an event-based manner, each service execution will fall into a unique box in the array. If a least one invocation occurs during the interval of time that is represented by a box, the box is shaded gray. The number of arrows pointing at each box is the number of consumers that are sharing the same service execution. Thus, the more arrows pointing at a box, the greater the degree of sharing, and the more energy is saved.

Figure 6.3(a) shows the service utilization when there are two consumers, C_1 and C_2, invoking at periods $P_1 = 4$ and $P_2 = 6$, respectively. C_1 thus executes the service at times 4, 8 and 12, as indicated by the blue arrows, while C_2 executes the service at times 6 and 12, as indicated by the green arrows. Note that the length of the array is equal to the least common multiple of 4 and 6 because beyond this, the invocation pattern repeats. Thus, service utilization can be calculated by only considering the block of times leading up to the least common multiple.

Calculating service utilization involves dividing the number of shaded boxes by the total number of boxes, which in this case is $\frac{4}{12} = \frac{1}{3}$. Figure 6.3(b) shows the utilization when a new consumer, C_3, invoking with period $P_3 = 2$, arrives. With this additional consumer, the new utilization is $\frac{1}{2}$, representing an increase of $\frac{1}{2} - \frac{1}{3} = \frac{1}{6}$. Note that this is less than an increase of $\frac{1}{2}$, which would be the case if service invocations could not be shared, further demonstrating the benefits of service sharing.

The visual method of calculating service utilization shown in Figure 6.3 depicts a relatively simple scenario in which the least common multiple is small. Unfortunately, aside from trivial simplifications like identifying periods that are multiples of each other or a period of 1, calculating the utilization of a service in general is complex.

For example, consider a naïve brute-force method for calculating utilization shown in Figure 6.4. Each position in time up to the least common multiple of all periods, lcm, is compared against each period. If at least one period evenly divides into the time interval being considered, an invocation occurs and a counter for the time interval is incremented. Thus, after considering every time interval up to lcm, the utilization is

```
1.  Given n periods: P_1, P_2, ..., P_n;
2.  Let lcm = Least Common Multiple of P_1, P_2, ..., P_n;
3.  Let count = 0;
4.  for i = 0 to lcm − 1
5.      for j = 1 to n
6.          if (i mod P_j) == 0 do
7.              count++;
8.              break; // out of inner for-loop
9.          end do
10. utilization = count/lcm
```

Figure 6.4: A naïve brute-force method for calculating utilization.

the count divided by lcm. Since each position in time from zero to lcm is compared against each of the n periods, the computational time complexity is $O(lcm \cdot n)$, which is exponential in the number of periods. The memory complexity is $O(1)$ since there is only one variable, *count*, being maintained.

A key problem with the algorithm presented in Figure 6.4 is the fact that it considers *every* interval of time from zero to lcm regardless of whether an invocation actually occurs at that point in time. If the service executions are sparse, there will be long stretches in time in which no executions occur. An algorithm that only considers the time intervals where executions occur is shown in Figure 6.5. It maintains a sorted list, *list*, that initially contains each service invocation period, P_1, P_2, ..., P_n. This initial value is the "base amount" that is continuously added to itself until it reaches lcm. With each round, the list is sorted and, if the smallest values are less than the least common multiple, they are incremented by their base amount. This process repeats until all values in *list* equal lcm. The number of rounds in the algorithm is equal to the number of positions in the timeline in which a service execution occurs, meaning the utilization is the number of rounds divided by lcm. The time complexity of this algorithm is $O(lcm \cdot utilization \cdot n \cdot log(n))$, which is exponential in the number of invocations. However, it is proportional to the utilization, which is usually small in WSNs, and the number of consumers is also expected to be small due to the limited wireless range of WSN nodes, meaning this algorithm is feasible in most situations. The memory complexity is $O(n)$ since it only needs to remember *list*.

In the current implementation, the savings achieved through service sharing is incorporated by the provider into P_{invoke} and $E_{tx,p}$, which are included in the response to

1. Given n periods: P_1, P_2, ..., P_n;
2. Let lcm = Least Common Multiple of P_1, P_2, ..., P_n;
3. Let $list = [P_1, P_2, \ldots, P_n]$;
4. Let $count = 0$;
5. sort($list$);
6. while smallest value(s) in $list$ are less lcm
7. increment smallest value(s) in $list$ by base amount;
8. sort($list$);
9. count++;
10. $utilization = count/lcm$

Figure 6.5: A algorithm for calculating service utilization when service sharing is possible.

a service discovery message. For example, if adding a consumer results in no change in the utilization of the service, and the results can be delivered via broadcast, then $P_{invoke} = 0$ and $E_{tx,p} = 0$ for that consumer. This results in consumers being biased towards providers that are better able to share service executions and thus save energy. One limitation to this approach is that it does not account for future changes to the set of bound consumers. To account for this, the provider can notify its consumers that the degree of sharing has decreased.

6.4.3 Adapting to Network Topology Changes

The mechanism for adapting to network topology changes runs on the consumer and is responsible for automatically switching providers to enhance service availability. It is necessary due to the transient connectivity between nodes in a WSN. As shown in Figure 6.6, the adaptation mechanism has only four states, imposing minimal overhead. The system maintains a list of known providers in a provider list, and a count of the number of consecutive failures using the providers in the list. The system begins in the `Init` state in which the provider list is empty. From this state, the system instantly transitions to the `Collect Providers` state while transmitting a service discovery message and setting timer T_{wait}, which controls the amount of time the consumer waits for matching providers to respond.

The service discovery message contains the specification of the required service and attributes indicating how the service is going to be invoked. That is, it specifies whether

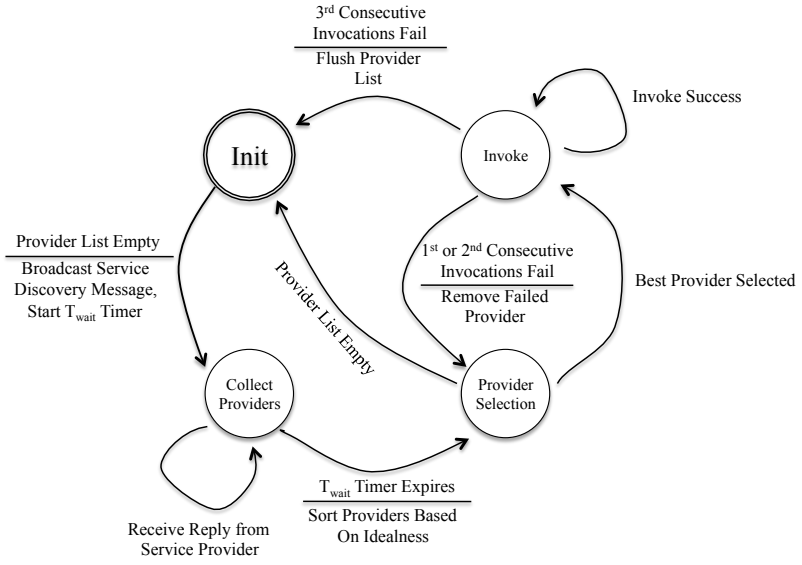


Figure 6.6: A finite state machine capturing the behavior of the adaptation mechanism used to adapt to network topology changes.

the service will be invoked on-demand, periodically, or in an event-based manner. If invoked periodically, it also specifies the period and duration of the invocations. If invoked in an event-based manner, only the period is specified. The contents of the service discovery message are used by the service provider to determine whether it is able to provide the necessary service and if so calculate the energy footprint the consumer will have on the provider. Assuming the provider is a match, the energy footprint is calculated and sent to the consumer. The consumer records this information in its provider list, and uses it to select the "best" provider, which is by default the one with the smallest energy footprint. The actual criteria for determining which provider is best can be customized via pluggable software modules.

After broadcasting the service discovery message, the consumer remains in the `Collect Providers` state accepting and recording responses from service providers until T_{wait} expires. When this occurs, the consumer sorts the list based on the aforementioned criteria for selecting the best provider, and enters the `Provider Selection` state. From this state, the consumer either selects the best provider and transitions into the `Invoke` state, or transitions back into the `Init` state if the provider list is empty.

Once in the `Invoke` state, the consumer invokes the service while remaining in the same state so long as the invocation remains successful. If the invocation fails, the current provider is discarded and the system returns to the `Provider Selection` state where it selects the next-best provider. This process of switching providers can repeat up to N consecutive times before the consumer gives up by flushing the provider list and returning to the `Init state`. The reasoning behind this is that N consecutive failures is indicative of a major change in network topology, e.g., when the consumer moves out of range of *all* previous providers. When this happens, the most logical action is to clear the provider list and re-discover new providers. The value N is exposed as a tunable parameter. It reflects the expected reliability of receiving a response from a provider, assuming one exists.

The entire adaptation mechanism shown in Figure 6.6 is performed by the middleware in a manner hidden from the application, except for a few tunable parameters. Specifically, the adaptive SOC middleware allows the developer to specify the algorithm for determining which provider is best and the values of T_{wait} and N. By presenting such a simple interface, application development is simplified.

The method of detecting invocation failure differs depending on the type of invocation being performed. On-demand invocations fail if a provider does not respond within a certain amount of time after an invoke message is sent. Periodic invocations fail if the consumer does not receive the expected number of invocation results at the requested frequency. Event-based invocations fail if the system does not continue to send interesting events back to the consumer. This can be detected when the current provider is removed from the neighbor list, which is maintained by lower-level services like a link estimator [54].

One important aspect of the adaptation mechanism is the fact that it is *reactive*. That is, it does not actively seek to change providers so long as the current provider remains available. The reasoning behind this is the fact that energy efficiency is of paramount importance to most WSN nodes, and needlessly searching for new providers when the current one is still available wastes energy. In addition, there is no guarantee that a more efficient provider exists, so proactively searching for another provider when the current one is available is risky in terms of waisting energy. Finally, some applications like habitat monitoring may infrequently invoke services. In this

case, proactive adaptation is wasteful if the application doesn't invoke the service between multiple adaptations. For these reasons, a passive mechanism that reacts to application invocations and provider disconnections is preferred.

6.5 Evaluation

The actual implementation of the adaptive SOA imposes minimal overhead in terms of memory and network bandwidth. On the TelosB, it consumes $20kb$ of ROM and $6.5kb$ of RAM, while on the Imote2, it consumes $187kb$ of ROM and $10kb$ of RAM. These are small relative to the amounts of memory available.

In terms of network bandwidth overhead, the adaptive SOA requires additional information related to energy efficiency to be included in certain messages. The service discovery message must contain four additional variables: the invocation type, period, and count, and whether the invocation results should be delivered reliably. This amounts to 8 bytes of data. The reply message to a service discovery must include six additional variables: $T_{tx,p}$, $E_{tx,p}$, $P_{idle,p}$, $E_{rx,p}$, T_{invoke}, and P_{invoke}. This amounts to 12 bytes of data and can easily fit within a single TinyOS packet. To support the adaptive SOA, the service specifications must include three additional variables: whether it is sharable, T_{invoke}, and P_{invoke}. This amounts to six bytes of data, and can also fit in a single packet.

The remainder of this section evaluates the additional burden placed on the application, service, and device developers in terms of what they must do to use the adaptive SOA presented in this chapter. Understanding the SOA's ease-of-use is important since a primary objective is to maintain usability and simplify application development. In the process, the values of the variables defined in Section 6.4 are derived. These variables will be used in Section 6.6.

Two types of nodes are examined in this evaluation: the Imote2 [37] and TelosB [144]. They represent two extremes in energy consumption among current WSN devices and are often used in today's WSNs. In addition, one service called `AccelTrigger` is evaluated. It involves sensing the accelerometer and is used by the structural health monitoring application discussed in Section 6.6.1. By deriving the properties of these

devices and service, a general idea of the burden placed on the application, service, and device developers is obtained.

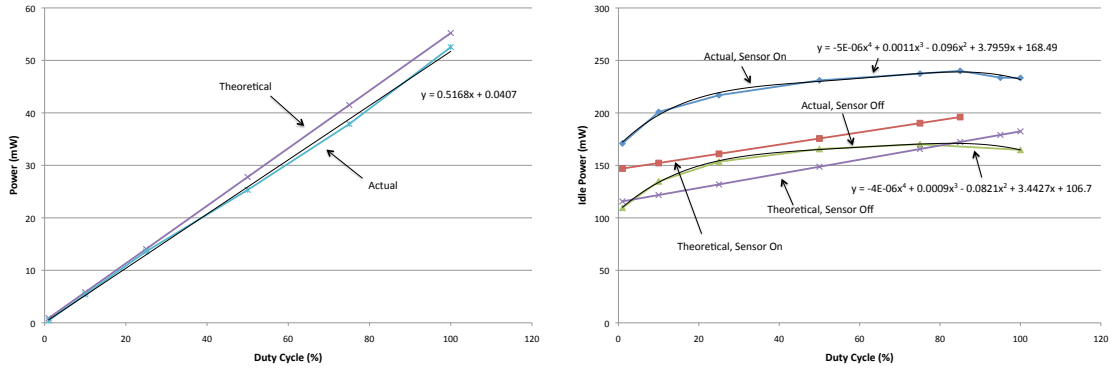
Using the adaptive SOA consists of determining the latency, power, and energy values listed in Table 6.1. Other parameters like `InvokePeriod`, `InvokeCount`, T_{wait}, whether a service sharable, and whether reliable results delivery is required, do not impose significant burden since they can be directly specified and do not require derivation. The remainder of this section analyzes how these values can be derived. It is divided in to four parts: the derivation of the variables associated with idling, sending, receiving, and sensing.

6.5.1 Energy Efficiency when Idling

Let P_{idle} be the amount of energy a device consumes when idle. A device is idle when it is not performing any application task like sending a message, taking a sensor reading, or performing computations. It is affected by `DutyCycle`, the duty cycle at which the radio operates since the radio continues to turn on and off even when the device is idle. To determine P_{idle} and how it is affected by `DutyCycle`, each device is attached to an oscilloscope and the power draw at various duty cycles is measured. For all measurements, the oscilloscope is set to sample at $250Hz$ ($4ms$ per sample) enabling the power draw measurements to be averaged over 10 second intervals. For details on how the oscilloscope is used to measure the energy consumption of a WSN device, see Appendix A.

The results for the TelosB node are shown in Figure 6.7(a). In addition to the actual measurement, a theoretical value is also plotted for comparison purposes. The theoretical value will be discussed later in this section. By fitting a linear trend line to the measured data, the idle power of the TelosB relative to duty cycle is given by equation 6.4.

$$P_{idle,telosb} \approx 0.5168 \cdot \texttt{DutyCycle} + 0.0407 \tag{6.4}$$



(a) TelosB with sensor board disabled (b) Imote2

Figure 6.7: Measured and theoretical P_{idle} of Imote2 and Telosb devices

The results for the Imote2 platform are shown in Figure 6.7(b). Two sets of measurements are shown: one when the sensor board is enabled, another when it is disabled. This is because when the Imote2 invokes the service locally, it must keep the sensor board enabled even when idling between service invocations. This is because driver limitations prevent disabling the sensor board between sensor readings. The TelosB, on the other hand, can turn its sensor board off when it is idle, thus saving energy. As mentioned previously, a theoretical value is also included for comparison purposes, which is discussed later in this section.

On the Imote2, the relationship between P_{idle} and `DutyCycle` is not linear. Instead it has a parabolic shape that is most likely due to latencies in switching on and off the radio on the Imote2 platform. Using a fourth-degree parabolic best-fit curve on the measured data, the equation for the Imote2's idle power when its sensor board disabled is as follows:

$$P_{idle,imote2,sensor\text{-}off} \approx -4 \cdot 10^{-6} \cdot \texttt{DutyCycle}^4 + 0.0009 \cdot \texttt{DutyCycle}^3 - 0.0821 \cdot \texttt{DutyCycle}^2 + 3.4427 \cdot \texttt{DutyCycle} + 106.7 \quad (6.5)$$

And when the sensor board is enabled, the idle power is:

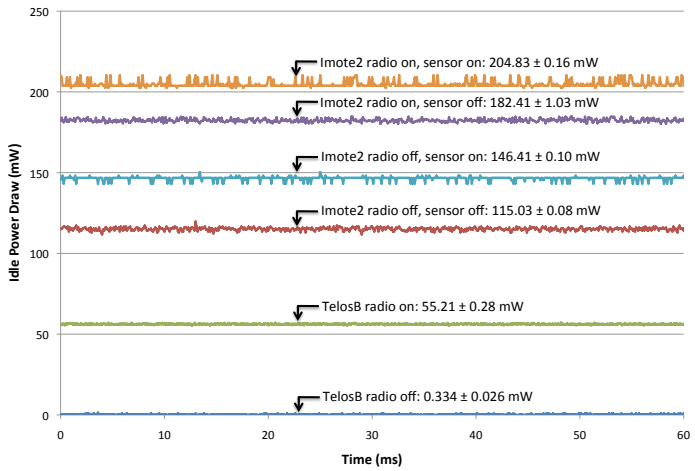


Figure 6.8: The power draw of the TelosB and Imote2 when idling with the radio on and off.

$$P_{idle,imote2,sensor\text{-}on} \approx -5 \cdot 10^{-6} \cdot \texttt{DutyCycle}^4 + 0.0011 \cdot \texttt{DutyCycle}^3 - 0.096 \cdot \texttt{DutyCycle}^2 + 3.7959 \cdot \texttt{DutyCycle} + 168.49 \tag{6.6}$$

Equations 6.4, 6.5, and 6.6 capture the actual P_{idle} of the TelosB and Imote2 platforms. Obtaining them is not difficult, as it only requires attaching the device to an oscilloscope and measuring their idle power draws with the radio set to various duty cycles. The developer of the device is responsible for determining P_{idle}. No additional burden is placed on the application programmer.

As previously mentioned, Figure 6.7 also includes the theoretical P_{idle} assuming the radio was duty cycled in an *ideal* manner. If the radio could be duty cycled in an ideal manner, P_{idle} should be a function of the power consumed when the radio is on, $P_{radio-on}$, and the power consumed when the radio is off, $P_{radio-off}$, as shown in equation 6.7.

$$P_{idle,ideal} \approx \frac{\texttt{DutyCycle}}{100} \cdot P_{radio-on} + \left(1 - \frac{\texttt{DutyCycle}}{100}\right) \cdot P_{radio-off} \tag{6.7}$$

However, due to hardware-specific properties like latencies when turning on and off the radio, the actual power consumed is different from the ideal. To determine how far off the theoretical value is relative to the actual value, $P_{radio-on}$ and $P_{radio-off}$ are measured using an oscilloscope. Figure 6.8 shows the amount of power drawn by the TelosB and Imote2 when the radio is on and off. Two measurements are given for the Imote2 depending on whether the sensor board is enabled. This is included because enabling the sensor board on the Imote2 consumes significant energy even when the node is idle. This is unlike the TelosB which can power down its sensor board between sensor readings. The results show that the TelosB consumes $0.334 \pm 0.026mW$ and $55.21 \pm 0.28mW$ when idle with the radio off and on, respectively. When the Imote2 sensor board is disabled, the Imote2 consumes $115.03 \pm 0.08mW$ and $182.41 \pm 1.03mW$ when the radio is off and on, respectively. When the sensor board is enabled, the Imote2 consumes $146.41 \pm 0.10mW$ and $204.83 \pm 0.16mW$ of power, respectively. Thus, the theoretical P_{idle} for the Imote2 when its sensor board is disabled is:

$$\begin{aligned} P_{idle,imote2,sensor\text{-}off,ideal} &\approx \frac{\texttt{DutyCycle}}{100} \cdot 182.41 + \left(1 - \frac{\texttt{DutyCycle}}{100}\right) \cdot 115.03 \\ &\approx 0.6738 \cdot \texttt{DutyCycle} + 115.03 \end{aligned} \tag{6.8}$$

When the Imote2's sensor board is enabled, it's theoretical P_{idle} is:

$$\begin{aligned} P_{idle,imote2,sensor\text{-}on,ideal} &\approx \frac{\texttt{DutyCycle}}{100} \cdot 204.83 + \left(1 - \frac{\texttt{DutyCycle}}{100}\right) \cdot 146.41 \\ &\approx 0.5842 \cdot \texttt{DutyCycle} + 146.41 \end{aligned} \tag{6.9}$$

And the theoretical P_{idle} for the TelosB is:

$$\begin{aligned} P_{idle,telosb,ideal} &\approx \frac{\texttt{DutyCycle}}{100} \cdot 55.21 + \left(1 - \frac{\texttt{DutyCycle}}{100}\right) \cdot 0.334 \\ &\approx 0.54876 \cdot \texttt{DutyCycle} + 0.334 \end{aligned} \tag{6.10}$$

Comparing the actual P_{idle} (Equations 6.5, 6.6, and 6.4) to the theoretical P_{idle} (Equations 6.8, 6.9, and 6.10), it is clear that the TelosB follows closely to the ideal, while the Imote2 does not. Thus, P_{idle} should be measured directly.

6.5.2 Energy Efficiency of Wireless Transmission

This section describes how to determine the amount of energy a device consumes when transmitting a message. It is used for specifying $E_{tx,c}$, $E_{tx,p}$, and $T_{tx,p}$ in Table 6.1. The key factors that influence the amount of energy consumed during message transmissions are: (1) the number of messages transmitted, (2) the amount of network bandwidth available, (3) the reliability of the wireless link over which the transmission is occurring, and (4) the type of duty cycling employed.

For the purpose of this analysis, the following two simplifying assumptions are made. First, a message consisting of one packet is used for all message exchanges, i.e., the initiation of the service invocation, and the delivery of the results, can all be done using one message. This assumption can later be removed by having the service specify another parameter – the number of messages necessary to invoke it and return the results. The second assumption is that there are no message retransmissions. This can be removed by having the devices specify the expected number of transmissions necessary to successfully send a packet over a particular link, which can be done using a variety of link estimators [54]. Both assumptions are not intrinsic to the adaptive SOA middleware model, and can be later removed.

The media access control (MAC) protocol has a significant impact on the efficiency of wireless transmission. In TinyOS 2.1, the default MAC layer is called BoxMAC-2 [124]. It uses *asynchronous duty cycling*, which reduces overhead by not using a global clock to synchronize the duty cycling of all nodes in the network. It results in a node having to first synchronize with the receiver before it can transmit a message. This is done by having the sender wait for the receiver to wake up before transmitting the message. Figure 6.9 shows the power draw of sending a message consisting of five packets when a 1% duty cycle is used. It was obtained using an oscilloscope and shows that there are three distinct stages to message transmission: search, send, and wait. The search stage consists of the device continuously retransmitting the first

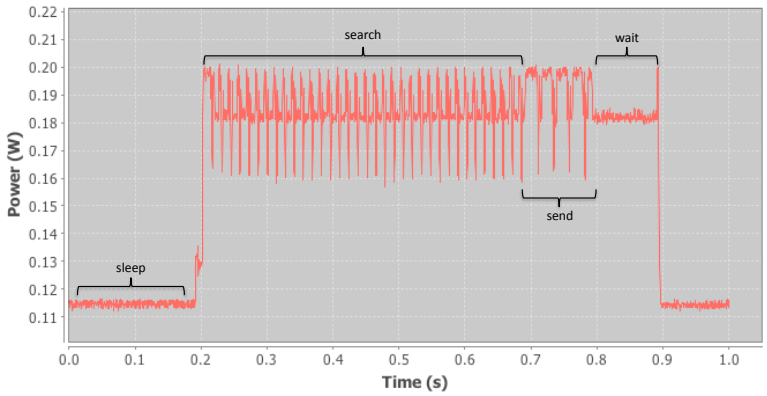


Figure 6.9: The power draw of an Imote2 when it transmits 5 packets.

packet until it is acknowledged by the receiver. This synchronizes the sender with the receiver. The second stage, send, consists of sending the four remaining messages. The last stage, wait, notifies the receiver of the end of the transmission.

For the remainder of this analysis, let E_i, T_i and P_i be the energy, latency, and power draw of performing task i. Using this nomenclature, the goal is then to find E_{tx}, the energy consumed during transmission, and T_{tx}, the transmission latency. From the sequence of steps for message transmission shown in Figure 6.9, equations 6.11 and 6.12 are derived.

$$E_{tx} = P_{search} \cdot T_{search} + P_{send} \cdot T_{send} + P_{wait} \cdot T_{wait} \tag{6.11}$$

$$T_{tx} = T_{search} + T_{send} + T_{wait} \tag{6.12}$$

Among the time variables used in equations 6.11 and 6.12, the only one that is dependent on the duty cycle is T_{search}, which has a range of 0 to `DutyCycle`. This is because in a best-case scenario the receiver will have its radio on when the sender initiates the search phase, and in the worse case the sender must wait an entire duty cycle before the receiver turns on. The other time variables are not dependent on `DutyCycle`. Specifically, T_{send} is a function of the network bandwidth since the

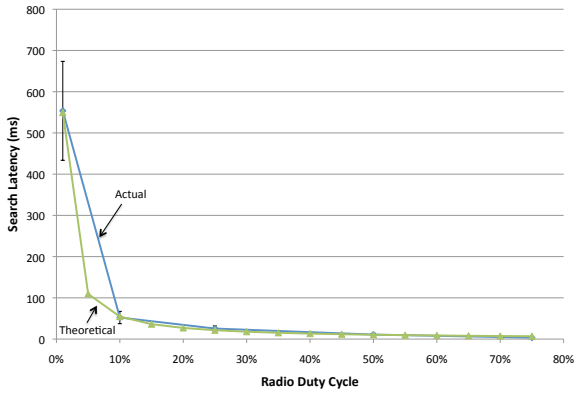


Figure 6.10: T_{search} versus the duty cycle, both actual and theoretical. The results indicate that, on average, T_{search} is half of the duty cycle period.

search phase ensures that the radios of both the sender and receiver are on, and T_{wait} is hard-coded into the MAC layer.[5]

Assuming a normal distribution of radio wake up times among nodes in a WSN, the average T_{search} is given by Equation 6.13.

$$T_{search} = \frac{\texttt{DutyCycle}}{2} \tag{6.13}$$

To determine whether Equation 6.13 is true, the following experiment is performed. A WSN device is configured to periodically send a packet. Each time a packet is sent, T_{search} is recorded using an oscilloscope. After collecting data from ten packet transmissions, the radio duty cycle is changed. The results of this experiment, along with the theoretical value derived from equation 6.13, are shown in Figure 6.10.[6] The results show that equation 6.13 is a valid characterization of T_{search} in the current system configuration.

The remaining variables in equations 6.11 and 6.12 can be measured using the same technique as that used in to evaluate T_{search}. Specifically, a node is configured to

[5]See `$TOSROOT/tos/chips/cc2420/lpl/DefaultLpl.h`, constant `DELAY_AFTER_RECEIVE`.

[6]In TinyOS 2.1, the equation for deriving the duty cycle period given the duty cycle is: $\texttt{DutyCyclePeriod (ms)} = \frac{11 \cdot (10000 - \texttt{DutyCycle} \cdot 100)}{\texttt{DutyCycle} \cdot 100} + 10$, where `DutyCycle` is expressed in percent form. For example, when the duty cycle is 50%, the period is $11 \cdot \frac{11 \cdot (10000 - 50 \cdot 100)}{50 \cdot 100} + 10 = 21ms$. For the actual code, see `$TOSROOT/tos/chips/cc2420/lpl/DefaultLplP.nc`, command `LowPowerListening.dutyCycleToSleepInterval(...)`

Variable	TelosB	Imote2	Unit
P_{search}	51.49 ± 0.11	184.44 ± 0.24	mW
P_{send}	0	0	mW
T_{send}	0	0	ms
P_{wait}	54.56 ± 0.06	182.81 ± 0.29	mW
T_{wait}	78.43 ± 2.59	86.5 ± 1.71	ms

Table 6.2: The timing and power attributes of sending one acknowledged packet. The numbers are obtained using an oscilloscope and averaged over ten packet transmissions. The average and 95% confidence intervals are shown.

periodically transmit a packet, and an oscilloscope is used to record the power draw and timing of the operation. Table 6.2 contains the results of the measurements. Note that P_{send} and T_{send} are both zero. This is because only one packet is being transmitted, and it is repeatedly transmitted during the search phase, which by definition ends when the first packet is delivered.

By plugging in the values of Table 6.2 and equation 6.13 into equations 6.11 and 6.12, the following equations for E_{tx} and T_{tx} are obtained:

$$\begin{aligned} E_{tx,imote2} &= P_{search} \cdot T_{search} + P_{send} \cdot T_{send} + P_{wait} \cdot T_{wait} \\ &= 184.44 \cdot \frac{\texttt{DutyCyclePeriod}}{2} + 0 \cdot 0 + 182.81 \cdot 86.5 \\ &= 184.44 \cdot \frac{\texttt{DutyCyclePeriod}}{2} + 15813.1 \end{aligned} \tag{6.14}$$

$$\begin{aligned} E_{tx,telosb} &= 51.49 \cdot \frac{\texttt{DutyCyclePeriod}}{2} + 0 \cdot 0 + 54.56 \cdot 78.43 \\ &= 51.49 \cdot \frac{\texttt{DutyCyclePeriod}}{2} + 4279.14 \end{aligned} \tag{6.15}$$

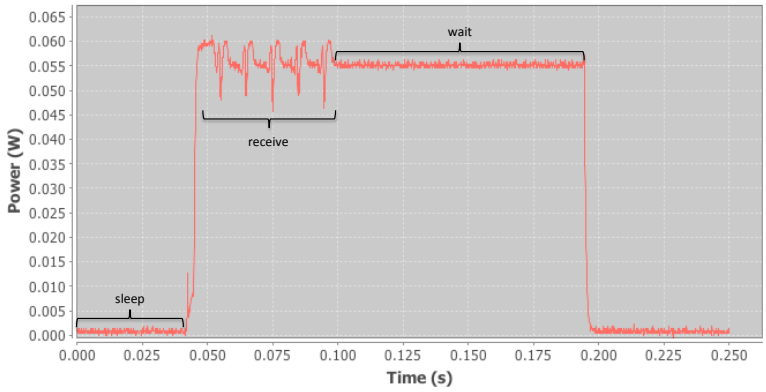


Figure 6.11: The power draw of a TelosB receiving 5 packets.

$$T_{tx,imote2} = T_{search} + T_{send} + T_{wait} \tag{6.16}$$
$$= \frac{\texttt{DutyCyclePeriod}}{2} + 86.5$$

$$T_{tx,telosb} = \frac{\texttt{DutyCyclePeriod}}{2} + 78.43 \tag{6.17}$$

The validity of Equations 6.14, 6.15, 6.16, and 6.17 will be established in the application case studies presented in Section 6.6, which use them to select energy efficient service providers. The task of obtaining equations for E_{tx} and T_{tx} is the responsibility of the device developer. It does not impose additional burden on the service or device developers.

6.5.3 Energy Efficiency of Wireless Reception

Figure 6.11 shows the power draw of a TelosB device during the reception of a 5-packet message when the radio duty cycle is set to 1%. Based on the figure, the receive operation can be divided into two parts: (1) *receive*, and (2) *wait*. The receive step is when the packets are actually be received. The *wait* step is a length of time the consumer remains awake after the reception of the last packet to ensure no additional

Variable	TelosB	Imote2	Unit
$P_{receive}$	53.63 ± 0.60	194.18 ± 0.38	mW
$T_{receive}$	16.45 ± 1.51	22.99 ± 2.13	ms
P_{wait}	54.03 ± 0.09	182.22 ± 0.15	mW
T_{wait}	93.25 ± 0.45	99.64 ± 0.33	ms

Table 6.3: The latency and power attributes of receiving a packet.

packets are destined for it. Thus, the following equations can be created for E_{rx} and T_{rx}, the energy and latency of reception, respectively.

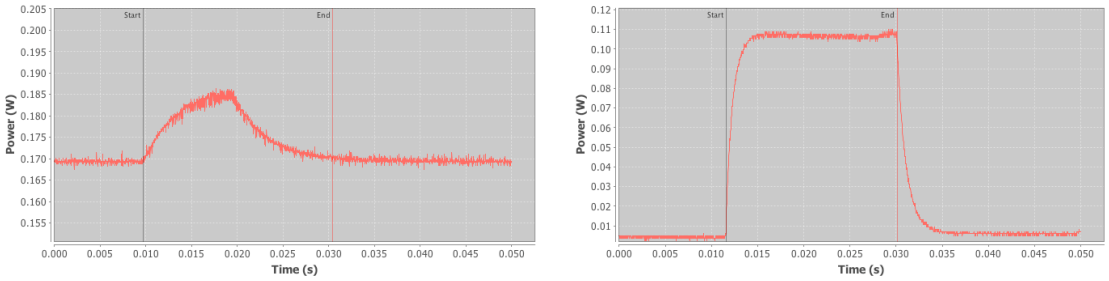
$$E_{rx} = (P_{receive} \cdot T_{receive}) + (P_{wait} \cdot T_{wait}) \tag{6.18}$$

$$T_{rx} = T_{receive} + T_{wait} \tag{6.19}$$

Using an oscilloscope, the power draws and latencies of the Imote2 and TelosB receiving a message can be measured. The results are shown in Table 6.3. By plugging in the values shown in Table 6.3 into equations 6.18 and 6.19, the energy cost and latency of message reception is obtained. The results are given in Equations 6.20, 6.21, 6.22, and 6.23.

$$\begin{aligned} E_{rx,imote2} &= (P_{receive} \cdot T_{receive}) + (P_{wait} \cdot T_{wait}) \\ &= 194.18 \cdot 22.99 + 182.22 \cdot 99.64 \\ &= 22620.6 \end{aligned} \tag{6.20}$$

$$\begin{aligned} E_{rx,telosb} &= 53.63 \cdot 16.45 + 54.03 \cdot 93.25 \\ &= 5920.51 \end{aligned} \tag{6.21}$$



(a) Imote2 (b) TelosB

Figure 6.12: The power draw of taking an accelerometer reading.

$$\begin{aligned} T_{rx,imote2} &= T_{receive} + T_{wait} \\ &= 22.99 + 99.64 \\ &= 122.63 \end{aligned} \tag{6.22}$$

$$\begin{aligned} T_{rx,telosb} &= 16.45 + 93.25 \\ &= 109.7 \end{aligned} \tag{6.23}$$

This task of obtaining equations for E_{rx} and T_{rx} is the responsibility of the device developer. It does not affect the service or device developers.

6.5.4 Energy Efficiency of Sensing

Recall from Section 5.7 that the `AccelTrigger` service is used to detect whether there is potential damage, and as a low-power monitoring state for the damage localization application. Each time the service is invoked, an acceleration reading is obtained, and the value is compared against a service-specific threshold. If the threshold is exceeded, the invocation is considered "interesting" and, assuming the service was invoked in an event-based manner, an event is signaled to the consumer.

The actual energy cost and latency of accessing the accelerometer sensor can be obtained using the oscilloscope and the technique described at the beginning of this

Variable	TelosB	Imote2	Unit
P_{sense}	102.90 ± 0.31	176.67 ± 0.45	mW
T_{sense}	18.49 ± 0.07	21.87 ± 1.28	ms

Table 6.4: The timing and power attributes of sensing.

Section. Two example oscilloscope traces, one for the Imote2, another for the TelosB, are shown in Figure 6.12. From the figure, the sensing operation can be represented in a single phase, thus let P_{sense} be the average power while sensing, and T_{sense} be the average latency of sensing. Using the oscilloscope, the accelerometer is accessed ten times and the average power and latency are computed. The results are shown in Table 6.4.

Since the sensing operation represents the vast majority of the cost of providing the AccelTrigger service (i.e., the computation to determine threshold is insignificant), the values shown in Table 6.4 are equal to P_{invoke} and T_{invoke}. In other words, $P_{invoke} = P_{sense}$ and $T_{invoke} = T_{sense}$.

This section has described how every variable in table 6.1 on page 140 except `InvokePeriod` and `InvokeCount` can be derived using an oscilloscope and a series of experiments. The two remaining variables are defined by the application developer. This section demonstrates that the additional attributes necessary for enabling energy-awareness can be obtained and that the process is feasible.

6.6 Applications

This section evaluates the adaptive SOA presented in this chapter using two application case studies: medical patient monitoring and structural health monitoring. They are used to evaluate the efficacy of our adaptation mechanism in terms of adjusting to network topology changes and increasing energy efficiency. Specifically, the medical patient monitoring application focuses on the ability to adapt to changing network topologies, while the structural health monitoring application focuses on energy-awareness.

Figure 6.13: A map of the WSN testbed used in the medical patient monitoring application. The testbed nodes provide relay services for delivering medical patient data to the base station, which is represented as a red triangle. The dotted lines marks the $358.71m$ route the patient traveled during each experimental round.

6.6.1 Medical Patient Monitoring

The medical patient monitoring application consists of a mobile user (patient) wearing a WSN device that monitors vital signs and periodically delivering the data to a central monitoring station. The delivery of the data from the patient to the base station is done via a fixed WSN infrastructure consisting of relay nodes embedded within the hospital building. As the patient moves, the set of relay nodes within range of the patient changes, requiring that the monitoring device adapt to the changing network topology. If it fails to adapt, critical patient data may not be delivered, jeopardizing the patient's life. The following evaluation determines how well the adaptive SOA presented in this chapter adapts to the changing network topology.

A similar clinical monitoring system has been deployed at the Barns and Jewish Hospital in St. Louis, and a clinical trial with real patients is currently underway. While the system deployed in the hospital was implemented in native nesC, reimplementing the system using the adaptive SOA demonstrated the efficacy of the simple programming model enabled by the middleware.

For this evaluation, the WSN testbed at Washington University in St. Louis [175] serves as the relay network for delivering patient data to the base station. It consists

of 73 TelosB nodes and spans the 5th floors of Jolley and Bryan Halls. A map of the testbed is shown in Figure 6.13. Each node in this network is line-powered, meaning they are *not* energy-constrained. Other than to provide power, the back-channel is used solely for debugging. For this evaluation, the radio power was set to 4 ($\sim$ -20dBm) for all experiments.

Within the relay network, the delivery of patient data is done using the Collection Tree Protocol (CTP) [59]. CTP is a many-to-one routing protocol for delivering data across a multi-hop network to a central base station that is included with TinyOS 2.1. Within the relay network used in this evaluation, it exhibited reasonably high reliability delivering messages from any node in the network to the base station with a reliability upwards of 90%. Given this relay network, the primary responsibility of the adaptive SOA is to successfully deliver *all* patient data to a relay node in the WSN testbed infrastructure. This is because, once delivered, CTP is responsible for delivering the data to the base station. To integrate CTP's relaying service with the adaptive SOA, CTP's interface is exposed as a service that is provided by each relay node. In all experiments, the medical patient traversed a fixed $358.71m$ long path that is indicated by the dotted lines in Figure 6.13. To determine the effects of patient speed, two speeds of walking were used, a slow walk averaging 0.6755 ± 0.009 m/s, and a fast walk averaging 1.333 ± 0.03 m/s.

Programming the medical patient monitoring application is straightforward. It consists of a single loop in which the patient data is obtained, followed by a single line invoking the relay service. The adaptive SOC programming model hides the complexity of adapting to network topology changes, enabling the application to remain simple.

For a base-line comparison, the medical patient monitoring application was also implemented using just CTP. This represents a native implementation that involves no SOC. By default, CTP uses the 4-bit link estimator (4BLE) and an exponentially-decaying algorithm for determining beaconing frequency, both of which are included with TinyOS 2.1. For this evaluation, all default settings and configurations were used. Since CTP technically does not invoke a service, this study focuses on how reliably the patient node is able to send patient data to its parent, which is a relay

	Adaptive SOA	4BLE
Fast Walk	$100\% \pm 0\%$	$31.16\% \pm 7.6\%$
Slow Walk	$100\% \pm 0\%$	$40.47\% \pm 11.2\%$

Table 6.5: The success rate of service invocation of the medical patient monitoring application.

node. Since the 4BLE decide's CTP's parent, the remainder of this section compares our adaptive SOA to the 4BLE.

Both the 4BLE and adaptive SOA versions of the application were run using fast and slow walks along the path shown in Figure 6.13. While traversing this path, the medical patient's node would attempt to send patient vital sign information consisting of a single 28-byte packet to the base station every 15 seconds, which is sufficient for monitoring most vital signs [34]. Each experiment was run ten times, enabling the calculation of average statistics and 95% confidence intervals.

The success rates of the adaptive SOA and 4BLE implementations are shown in Table 6.5. In both the fast and slow walk scenarios, the adaptive SOA was able to maintain 100% success rate, while the 4BLE failed a significant percentage of times (its success rate was only $40.4 \pm 11.2\%$ and and $31.2 \pm 7.5\%$ for the slow and fast walking scenarios, respectively). The 4BLE performs poorly because it does not incorporate mechanisms for quickly adapting to network topology changes. Using the exponentially-decaying beaconing algorithm, after its initial rapid broadcasts of beacons, its latency of discovering new relay nodes increases into the range of several-minutes, with a maximum of 8.5 minutes. While this is acceptable in a stable (i.e, non-mobile) network, for which the 4BLE was originally intended, it is not acceptable when mobile nodes are involved since the set of nodes that are within wireless range of the mobile node change faster than the 4BLE is able to discover them. The adaptive SOA clearly outperforms the 4BLE, demonstrating the need to adapt to changing network topologies, and the efficacy of the middleware's adaptation mechanism.

In addition to success rate, consider the network bandwidth overhead as defined by the number of packets transmitted by the patient's device per service invocation. The non-beacon portion of the network bandwidth overhead is shown in Figure 6.14. The overhead imposed by beacons is discussed later in this section. The average and 95%

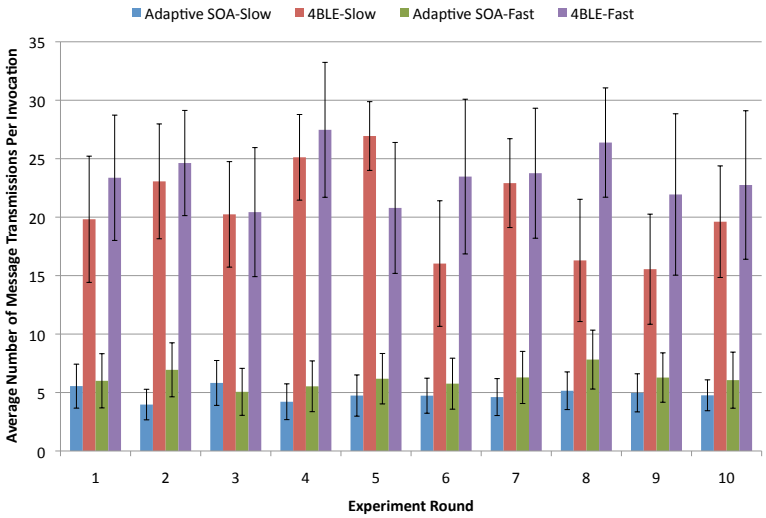


Figure 6.14: The average number of messages transmitted per invocation.

	Adaptive SOA	4BLE
Fast Walk	$0.79 \pm .03$	2.41 ± 0.80
Slow Walk	$0.47 \pm .05$	2.38 ± 0.55

Table 6.6: The average number of beacons transmitted per invocation over all experimental rounds.

confidence interval over 10 experimental rounds are shown. Note that the adaptive SOA out-performs the 4BLE transmitting less than ten packets per invocation while the 4BLE transmits about 15-25. This indicates that the adaptive SOA saves energy by transmitting fewer packets, while providing higher service availability.

The reason for the variance in the number of packets transmitted per invocation is due to the changes in connectivity to service providers. For example, if the original service provider is still within range, the adaptive SOA may be able to invoke the service using a single packet transmission. Otherwise, it will have to perform service discovery, which consists of broadcasting a 5 packets. The 4BLE attempts to send the message to the parent up to 30 times, after which it drops the packet. It does not attempt to discover different parents if the currently-selected one fails. Thus, the number of messages it sends is between 1 and 30, depending on whether the parent is within range.

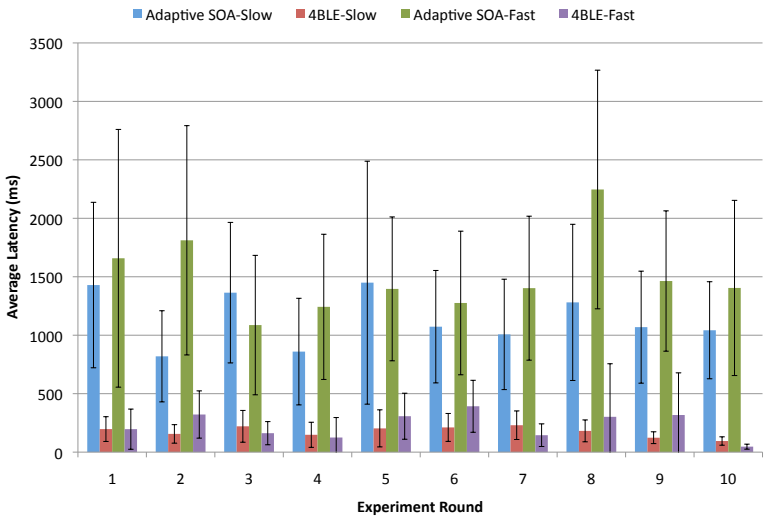


Figure 6.15: The latency of invoking the relay service.

The average number of beacons emitted per invocation is shown in Table 6.6. Clearly, the 4BLE emits many more beacons than the adaptive SOA, while delivering lower success rate. The 4BLE emits more beacons because it uses the link estimator for discovering the parent, which rapidly re-broadcasts beacons whenever it detects dynamics in the network. As the patient node moves, the link estimator running on the node may detect changes in the network (based on beacons from new providers) resulting in additional beacons being emitted. More importantly, CTP tells the 4BLE every time it fails to invoke the service, causing the link estimator to emit beacons at a faster rate. The net result is the 4BLE sending about 1.77 additional beacons per service invocation relative to the adaptive SOA.

The average latency of invoking the relay service is shown in Figure 6.15. 95% confidence intervals are included based on the ten experimental rounds. The results indicate that the adaptive SOA has higher latency than the 4BLE. This is because the adaptive SOA has an adaptation mechanism that continuously retries the service invocation with different relay nodes until it succeeds. Since this process may take many rounds depending on whether any providers are within range, its latency may occasionally be high. However, from the application's perspective, the higher latency is justified by the 100% success rate of invoking services and lower network overhead provided by the adaptive SOA.

6.6.2 Structural Health Monitoring

Structural health monitoring (SHM) is a class of WSN applications that use WSNs to monitor the health of structures like buildings and bridges. A key challenge of SHM applications is the need to run for long periods of time, ideally for the life of the structure, despite having limited energy. The fact that most SHM algorithms are computationally heavy and energy intensive only magnifies the problem. To address this, one solution is to integrate energy-efficient nodes that simply monitor the vibrations in the building, and signals an event whenever the vibrations are large enough to result in structural damage. Using these low-power nodes, the energy-intensive algorithms do not have to continuously run on the high-powered nodes, thus saving energy.

While the application case study described in Chapter 5 demonstrated that this technique can save energy, the process of selecting the node that performs the low-power monitoring was done manually, and the system did not automatically determine the energy cost of selecting a particular node. This section presents how an adaptive SOA can improve on this technique by automatically determining the energy footprint of selecting a particular node. The results are validated by comparing the estimated energy consumption to the actual energy consumption.

The system configuration is as follows. There are two nodes in the network, an Imote2 and a TelosB. The Imote2 is a high-powered but energy-inefficient node that is both a consumer and provider, while the TelosB is a low-powered but energy-efficient node that is just a provider. Both nodes provide a service called `AccelTrigger`, which performs the low-power monitoring. As a consumer, the Imote2 must bind to and invoke the `AccelTrigger` service to save energy. Given this setup, there are two binding states: 1) the Imote2 can bind to the *AccelTrigger* service locally, or 2) it can bind to the service remotely by using the one provided by a TelosB. In addition, there are two variables that need to be supplied by the consumer, `InvokePeriod`, and `InvokeCount`, as specified in Table 6.1. The challenge, then, becomes how to determine the energy footprint in terms of the binding state, `InvokePeriod`, and `InvokeCount`.

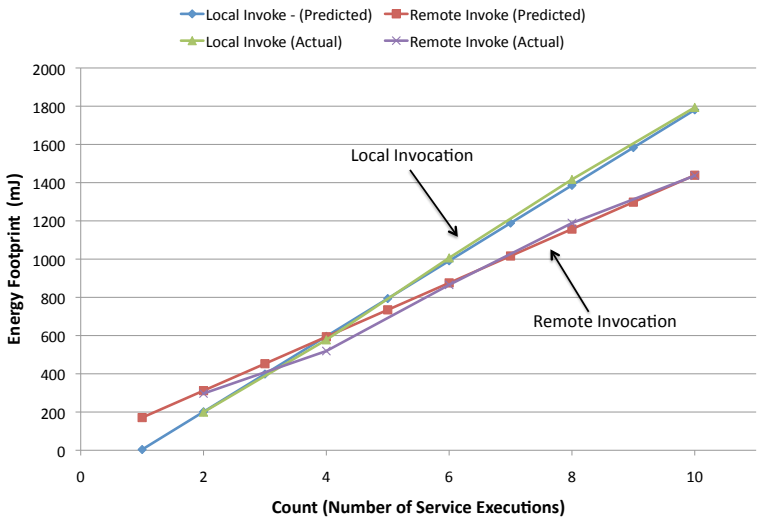


Figure 6.16: The predicted and actual energy footprints of the structural health monitoring application scenario when `DutyCycle` = 10 and `InvokePeriod` = 1000 in.

Predicting the energy footprint requires using equations 6.3 and 6.2 with values specific to the Imote2 and TelosB platforms, which were derived in Section 6.5. Assuming `DutyCycle` = 10 and `InvokePeriod` = 1000, the question is what is the energy footprint of each binding state relative to `InvokeCount`, and when are remote invocations more energy-efficient than local invocations? The results are shown in Figure 6.16. The actual values were obtained by directly measuring the energy consumption of the system using an oscilloscope. The predicted values were obtained using the values derived in Section 6.5, the details of which are given in Appendix B. Note that the predicted energy footprints closely match the actual energy footprints, and that both result in the same conclusion: that `InvokeCount` must be at least 4 for remote binding to be more energy efficient than local binding. Specifically, the measured intersection point between the energy footprint of local vs. remote binding is `InvokeCount` = 3.45, which is close to the predicted 3.95.

Implementing this application using the adaptive SOA is simple. It consists of a single call to invoke the `AccelTrigger` service, followed by a callback function implementing the normal energy-intensive structural health monitoring application. As intended, the process of selecting and binding to the most energy efficient provider is hidden from the application.

6.7 Chapter Summary

This chapter presented an adaptive service provisioning framework that enhances service availability and energy efficiency transparently from applications. The framework features three novel adaptation strategies specifically designed for service provisioning in WSNs: 1) energy-aware service selection, 2) opportunistic service sharing, and 3) adaptive service rebinding in response to network dynamics. Naturally incorporated into an SOC paradigm, the adaptive strategies are hidden from the device, service, and application developers and thereby simplify application development. Empirical results from implementations on TelosB and Imote2 platforms and an evaluation of two applications, medical patient monitoring and structural health monitoring, demonstrate the systems efficiency and efficacy.

Chapter 7

Future Work

The work presented in this dissertation represents the first steps towards achieving adaptive middleware for resource-constrained mobile ad hoc and wireless sensor networks. There is still much additional work to be done. Clearly, additional application case studies can be run to better evaluate the middleware systems presented in this dissertation. More work needs to be done testing, validating, and improving the power equations used in Section 6.4. For example, the equations can be improved by accounting for messages of different sizes and network link qualities to vary. The semantics of the service invocation operations can be further developed. For example, perhaps they can be modified to provide "group" operations in which multiple identical services are redundantly invoked to increase reliability. The current evaluations primarily focus on two types of devices: the TelosB and Imote2. In the future, a wider range of devices must be analyzed to verify the energy equations apply to a wider set of devices. In addition, all of the work done on WSNs used TinyOS as the underlying operating system. While TinyOS is sufficient, there are many other operating systems for WSNs that provide different services like dynamic memory. A direction of future work involves investigating how the operating system affects the programming models and their implementations described in this dissertation. These are just some examples of the ways in which the existing middleware systems can be improved. Many other possibilities exist.

One area of future work is the development of novel coordination middleware in to address challenges unique to new forms networks and application domains. For example, body sensor networks and augmented reality applications are rapidly evolving and will require the development of middleware to assist application development.

They exhibit different characteristics they may require the middleware to be tailored to their intricacies. For example, body sensor networks are deployed on the human body instead of in an environment. Because of this, the behavior of the network links in terms of reliability and dynamics differs from that of traditional WSNs. Augmented reality involves overlaying virtual data on top of the physical world. They require high levels of context awareness and to be able to quickly detect changes in the context. Developing novel middleware systems that address these and other needs as they present themselves is a subject of future work.

Regarding Limone, a primary objective was to minimize the number of assumptions made about the behavior of the underlying network. It achieved this by providing a lightweight coordination model that does not require distributed transactions or predictable network connectivity. The result is a coordination model that does not provide any guarantees regarding the success of an operation. While this is sufficient for certain non-critical applications, it is insufficient for others. Other coordination models like Limone do provide additional guarantees about atomicity and more powerful group operations, but fail to function in highly dynamic networks that do not meet the underlying assumptions of the coordination model. Developing a hybrid coordination model that is able to adapt to degree of unpredictability in the network and application demands is a subject of future work. For example, if the network is behaving unpredictability and the applications do not require certain guarantees on the successful execution of certain operations, the middleware can adapt to provide the minimal level of service required to save resources. Identifying the interface that is expressive enough to enable this variability in middleware functionality is the subject of future work.

Nearly all coordination models to date assume that network links are bidirectional. This is made possible by implementing a layer of software that filters out asymmetric links, in which messages can be transmitted in one direction only. An interesting avenue of future work is to investigate middleware that can make use of asymmetric links, perhaps by using them to increase the communication bandwidth between two points in the network. Determining the proper high-level abstraction for enabling applications to make use of these asymmetric links is the subject of future work.

A key contribution of Servilla is the modularity of the middleware that enables the inclusion of even extremely weak devices. The current implementation divides the service provisioning framework (SPF) into two relatively course modules: the SPF-Provider and SPF-Consumer. While this was sufficient to enable the inclusion of TelosB devices, there will certainly be devices that remain too weak to be included. The future research involves identifying novel ways in which the middleware can be decomposed or rearranged to enable support for even weaker devices. For example, the current architecture forces every SPF-Provider to publish the specifications of the services it provides. Suppose the device is so weak that it does not even have enough memory to hold the service specifications. Perhaps a 3rd party service registry can hold the service specifications, and vouch for these extremely resource-poor devices.

Another area of future work lies in the integration of the various middleware systems discussed in this dissertation. For example, perhaps Agilla can be integrated with Servilla. Agilla would enable applications to be self-adaptive by reconfiguring their code in response to changing network conditions. Servilla would enable applications to operate in WSNs consisting of heterogeneous devices, adapt to changing network topologies, and optimize for energy efficiency. Merging Agilla with Servilla would facilitate the development of self-adaptive applications in heterogeneous WSNs.

A phenomenon of potential interest was discovered while performing the energy measurement experiments for the adaptive SOA. Specifically, there are collateral costs associated with wireless transmission. Most systems today assume that when a packet is sent, the only nodes affected are the sender and receiver. While this is approximately true in traditional networks, extremely low-energy and low-duty cycle networks like those created by WSNs are effected in significant ways by collateral cost due to a node overhearing a packet that is not destined for it. In fact, for the cc2420 radios, which are used by the TelosB and Imote2 devices, the costs of overhearing a packet is equal to the cost of receiving it as it were the intended recipient. This is because cc2420 are packet level radios, meaning they must receive the whole packet before they can determine if they are the intended recipient. Collateral energy costs depends on the number of nodes within range of the sender, which varies across nodes and varies over time. More investigation is necessary to determine how these variable collateral energy costs can be quantitified and how existing energy-aware communication protocols are affected.

Chapter 8

Conclusions

This dissertation presented three middleware platforms, Limone, Agilla, and Servilla, that address different challenges present in mobile ad hoc networks (MANETs) and wireless sensor networks (WSNs). Limone is a lightweight coordination middleware for MANETs. It showed how lightweight coordination primitives are useful in facilitating the development of certain applications like a universal remote control even if they do not provide strong functional guarantees. Agilla is a middleware for WSNs that enables application developers to structure their applications as collections of mobile agents that communicate through localized tuple spaces. In doing so, Agilla enables applications to self-adapt by restructuring the locations of their code in response to changes in the environment, which reduces an application's overhead and increases the utility of the WSN by enabling additional applications to run. Example application case studies involving wildfire tracking and cargo container monitoring demonstrated how the Agilla middleware simplifies the development of complex applications by enabling the code to self-adapt. In addition, by combining Agilla with Limone, WSNs can be seamlessly integrated with traditional networks by enabling mobile agents to travel from one type of network into another, enabling computations to span networks. This was demonstrated through the development of the Agimone middleware system. Servilla is a middleware that address challenges due to network heterogeneity in dynamics within a WSN. It demonstrated that service-oriented computing (SOC) can be used to enable applications to be platform-independent while still able to access the full functionality of the underlying hardware. In addition, it showed how the middleware can exploit the decoupling of service consumers and providers established by the SOC programming model to enable applications to automatically adapt to changing network conditions and increase energy efficiency.

Accomplishing this required minimal additional effort from the developer, who only needs to specify a limited set of variables characterizing the energy efficiency of a device and service. This was demonstrated using two application case studies: medical patient monitoring and structural health monitoring. In all, the systems described in this dissertation represent the first steps towards providing adaptive middleware for resource-constrained MANETs and WSNs.

Appendix A

Measuring the Energy Consumption of WSN Devices

This appendix describes how the energy consumption of a WSN device is measured. Determining the amount of energy used by WSN devices is important due to the scarcity of energy available. Most WSN devices operate on batteries that cannot be easily recharged or replaced since the devices themselves are embedded within the environment.

Measuring the amount of energy a device consumes requires determining the power draw of the device and the duration over which the power is drawn. By taking a sequence of instantaneous power readings $P_1, P_2, \ldots, P_n$ at fixed intervals in which each interval is of length Δt, the total energy, E, consumed during the period measured is given by equation A.1.

$$E = \left(\sum_{i=1}^{n} P_i \right) \cdot \Delta t \tag{A.1}$$

The power draw of the WSN device can be determined by measuring the current draw and the voltage drop across the device. The circuit used to obtain these measurements is shown in Figure A.1. A high-accuracy resistor, $R1$, is put in parallel with the WSN device. By measuring the voltage across the resistor ($V1$), the current flowing through the device can be derived using the formula $I = \frac{V1}{R1}$. In addition to the voltage across the resistor, the voltage across the device, $V2$, is measured directly. Once these values are obtained, they can be used to compute the energy using equation A.1.

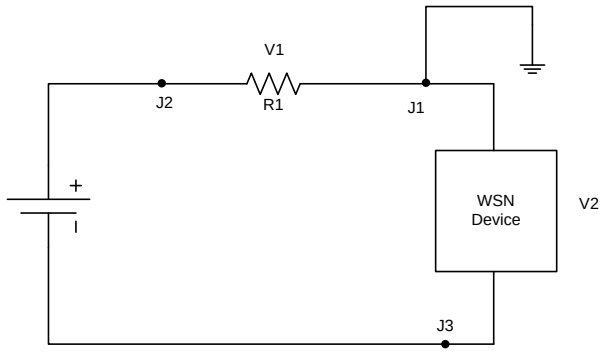


Figure A.1: The circuit used to measure the power draw of a WSN device. Two probes from the same oscilloscope simultaneously measure voltages $V1$ and $V2$ at junctions J2 and J3, respectively. Both are grounded at junction J1. $V1$ measures the voltage across resistor R1 and is used to calculate the instantaneous current, $i = \frac{V1}{R1}$. $V2$ measures the voltage across the WSN device. The power, P, of the WSN device is thus $P = i \cdot V2$

A Tektronix TDS 2004B digital oscilloscope is used to obtain $V1$ and $V2$ for all measurements presented in Chapter 6. The channel 1 probe is attached to junction $J2$, while the channel 2 probe is attached to $J3$. Both probes are grounded at $J1$ to provide a common point of reference. Thus, the channel 1 probe measures the voltage across resistor $R1$, which is then used to calculate the current flowing through the circuit. The channel 2 probe measures the voltage drop across the device. To derive the energy, the measurements are processed using a Java application that reads the raw measurements obtained from the oscilloscope. The oscilloscope's buffer allows capturing 2500 instantaneous voltage readings, each separated by an equal amount of time. Depending on the time-scale settings of the oscilloscope, the time between each measurement is anywhere between 0.01ms to 4ms. Since most operations being measured have latencies of tens of milliseconds, the resolution provided by the oscilloscope is sufficient in most scenarios.

Since the oscilloscope is limited in the number of data points it can measure, and the longer the period the coarser the energy calculations, the intervals over which energy utilization is calculated must be carefully selected. Naturally, the selection should be based on the variables that need to be measured to enable energy-aware service adaptation in Servilla, which are shown in Table 6.1. Some of the variables are actually functions of the duty cycle requiring each variable to be evaluated with the device set to various duty cycles.

Appendix B

Derivation of the Energy Utilization Equations for the Structural Health Monitoring AccelTrigger Service

This section describes how the middleware can automatically calculate the energy consumption of different states of the structural health monitoring application described in Section 6.6.2. The system consists of two nodes, an Imote2 and a TelosB. The Imote2 is both a service consumer and provider, while the TelosB is just a provider. The service used is called `AccelTrigger`. It periodically senses the accelerometer and sends the consumer an event whenever the acceleration reading exceeds a certain value. The purpose of using `AccelTrigger` is to save energy. Instead of continuously running the complex and energy-intensive damage localization task, the application can run `AccelTrigger` during periods in which it is not sure the building is damaged. Relative to the damage localization task, `AccelTrigger` consumes much less energy, thus motiving its use during quiescent periods.

The following analysis derives the equations used to determine whether the consumer, a high-powered Imote2 node, should invoke the service locally or remotely on a low-power TelosB node. The equations used are those described in Section 6.4, but integrated the device-specific equations and attributes collected from actual hardware, as described in Section 6.5. Using these equations, the consumer node can decide whether invoking locally or remotely is more energy efficient.

The remainder of this appendix is organized as follows. First, the energy cost of invoking locally will be presented. Second, the energy costs of invoking remotely will be presented. The third section calculates the potential energy savings, and what its ramifications are in terms of when invoking remotely is better than locally. The macro-level experiments on actual hardware that validate the conclusions of the algorithms presented in this appendix are described in Section 6.6.2.

B.1 Local Invocation

When the service is invoked locally, the Imote2 is invoking the service that it provides itself. Thus, only the energy incurred by the Imote2 is considered.

Energy Consumption on Imote2:

The basic equation for determining the total energy consumption on the Imote2 when it invokes the service locally is given by equation 6.2 on page 141. For convenience, it`InvokeCount` is repeated here.

$$E_{local} = \texttt{InvokeCount} \cdot T_{invoke} \cdot P_{invoke} + (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) \cdot P_{idle} \tag{B.1}$$

Since the Imote2 is invoking the service locally, it must have its sensor board on. Thus, P_{idle} is specified by equation 6.6 on page 152, and the values of P_{invoke} and T_{invoke} are given in Table 6.4 on page 161. The resulting equations are as follows:

$$T_{invoke} = 21.87 \tag{B.2}$$

$$P_{invoke} = 176.67 \tag{B.3}$$

$$P_{invoke} = -5 \cdot 10^{-6} \cdot \texttt{DutyCycle}^4 + 0.0011 \cdot \texttt{DutyCycle}^3 - 0.096 \cdot \texttt{DutyCycle}^2 + 3.7959 \cdot \texttt{DutyCycle} + 168.49 \tag{B.4}$$

The total energy cost is thus equations B.2 through B.4 plugged into equation B.1.

B.2 Remote Invocation

Since the `AccelTrigger` service is being invoked in an event-based manner, equation 6.1 on page 139 is used.

$$\begin{aligned} E_{remote,event} = E_{tx,c} + E_{rx,p} & \\ + \texttt{InvokeCount} \cdot (P_{idle,c} \cdot T_{invoke} + T_{invoke} \cdot P_{invoke}) & \\ + (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) & \\ \cdot (P_{idle,c} + P_{idle,p}) & \\ + E_{tx,p} + E_{rx,c} & \end{aligned} \tag{B.5}$$

where

$$\begin{aligned} E_{tx,c} &= E_{tx,imote2} \\ &= 184.44 \cdot \frac{\texttt{DutyCyclePeriod}}{2} + 15813.1 \end{aligned} \tag{B.6}$$

$$\begin{aligned} E_{tx,p} &= E_{tx,telosb} \\ &= 51.49 \cdot \frac{\texttt{DutyCyclePeriod}}{2} + 4279.14 \end{aligned} \tag{B.7}$$

$$\begin{aligned} E_{rx,c} &= E_{rx,imote2} \\ &= 22620.6 \end{aligned} \tag{B.8}$$

$$E_{rx,p} = E_{rx,telosb} \tag{B.9}$$
$$= 5920.51$$

$$P_{idle,c} = P_{idle,imote2,sensor\text{-}off} \tag{B.10}$$
$$= -4 \cdot 10^{-6} \cdot \texttt{DutyCycle}^4 + 0.0009 \cdot \texttt{DutyCycle}^3$$
$$- 0.0821 \cdot \texttt{DutyCycle}^2 + 3.4427 \cdot \texttt{DutyCycle} + 106.7$$

$$P_{idle,p} = P_{idle,telosb} = 0.5168 \cdot \texttt{DutyCycle} + 0.0407 \tag{B.11}$$

$$T_{invoke} = T_{sense,telosb} = 18.49 \tag{B.12}$$

$$P_{invoke} = P_{sense,telosb} = 102.90 \tag{B.13}$$

The total cost when invoking remotely is thus equations B.6 through B.13 plugged into equation B.5.

B.3 Local vs. Remote Invocation

To determine when invoking a remote service is better than invoking a local service, the difference in energy cost between local vs. remote invocation must be calculated. This is done by subtracting equation B.5 from B.1. That is,

$$E_{savings} = E_{local} - E_{remote,event} \tag{B.14}$$

Energy savings is possible when $E_{savings} > 0$, since that would imply that remote invocation is less costly than local invocation.

B.4 Example Scenario 1

Understanding when energy savings is possible is difficult when there are three variables (`DutyCycle`, `InvokePeriod`, and `InvokeCount`). Thus, to gain a better understanding of when energy savings is possible, the following is an example when the radio duty cycle and service invocation period are fixed. It derives the energy cost equations and generates a conclusion regarding the number of service invocations that must occur before a net energy-savings is achieved. This is what the middleware does when it makes a decision regarding whether to bind to a local or remote service. After deriving the costs and conclusion that Servilla would make, an actual system is evaluated and used to validate the decision.

B.4.1 Energy-Aware Calculations

Suppose the radio is operating on a 10% duty cycle (`DutyCycle` = 10), and the sensor is accessed once per second (`InvokePeriod` = 1000). The energy savings is calculated by plugging these values into equation B.14. This is done as follows. First, determine E_{local}, the energy cost when invoking the service locally.

$$\begin{aligned} E_{local} &= \texttt{InvokeCount} \cdot T_{invoke} \cdot P_{invoke} \\ &\quad + (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) \cdot P_{idle} \\ &= \texttt{InvokeCount} \cdot 21.87 \cdot 176.67 \\ &\quad + (\texttt{InvokeCount} - 1) \cdot (1000 - 21.87) \cdot 197.90 \\ &= 197436 \cdot \texttt{InvokeCount} - 193572 \end{aligned} \tag{B.15}$$

Next, determine E_{remote}, the energy cost when invoking the service remotely.

$$
\begin{aligned}
E_{remote,event} = (E_{tx,c} + E_{rx,p}) & \\
& + \texttt{InvokeCount} \cdot (P_{idle,c} \cdot T_{invoke} \\
& \quad + T_{invoke} \cdot P_{invoke}) \\
& + (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) \\
& \quad \cdot (P_{idle,c} + P_{idle,p}) \\
& + (E_{tx,p} + E_{rx,c})
\end{aligned}
\tag{B.16}
$$

Equation B.16 can be divided into two parts: (1) the energy associated with network communication, and (2) the energy associated with executing the service.

$$
E_{remote,event} = E_{network} + E_{execution} \tag{B.17}
$$

$$
E_{network} = (E_{tx,c} + E_{rx,p}) + (E_{tx,p} + E_{rx,c}) \tag{B.18}
$$

$$
\begin{aligned}
E_{execution} = \texttt{InvokeCount} \cdot (P_{idle,c} \cdot T_{invoke} + T_{invoke} \cdot P_{invoke}) & \\
+ (\texttt{InvokeCount} - 1) \cdot (\texttt{InvokePeriod} - T_{invoke}) & \\
\cdot (P_{idle,c} + P_{idle,p}) &
\end{aligned}
\tag{B.19}
$$

Deriving the equation for $E_{network}$ consists of plugging in the equations for the energy cost of transmission and reception given in Sections 6.5.2 and 6.5.3.

$$
\begin{aligned}
E_{network} &= \left(184.44 \cdot \frac{1000}{2} + 15813.1\right) + 5920.51 \\
&\quad + \left(51.49 \cdot \frac{1000}{2} + 4279.14\right) + 22620.6 \\
&= 166,598 \mu J
\end{aligned}
\tag{B.20}
$$

Deriving the equation for $E_{execution}$ requires first calculating $P_{idle,c}$ and $P_{idle,p}$. This is done by plugging in $\texttt{DutyCycle} = 10$ into equations 6.5 and 6.4.

$$P_{idle,c} = P_{idle,imote2,no\text{-}sensor} = 133.78\mu J$$
$$P_{idle,p} = P_{idle,telosb} = 5.3087\mu J$$

Thus,

$$\begin{aligned} E_{execution} &= \texttt{InvokeCount} \cdot (133.78 \cdot 18.49 + 18.49 \cdot 102.90) \\ &\quad + (\texttt{InvokeCount} - 1) \cdot (1000 - 18.49) \\ &\quad \cdot (133.78 + 5.3087) \\ &= 140893 \cdot \texttt{InvokeCount} - 136517 \end{aligned} \tag{B.21}$$

From equation B.17, adding equations B.20 and B.21 results in $E_{remote,event}$, the amount of energy consumed when the service is invoked remotely.

$$\begin{aligned} E_{remote,event} &= E_{network} + E_{execution} \\ &= 166598 + (140893 \cdot \texttt{InvokeCount} - 136517) \\ &= 140893 \cdot \texttt{InvokeCount} + 30081.1 \end{aligned} \tag{B.22}$$

Thus, the potential energy savings is equation B.22 subtracted from equation B.15, as given by equation B.14.

$$
\begin{aligned}
E_{savings} &= E_{local} - E_{remote,event} \\
&= (197436 \cdot \texttt{InvokeCount} - 193572) \\
&\quad - (140893 \cdot \texttt{InvokeCount} + 30081.1) \\
&= 56542.8 \cdot \texttt{InvokeCount} - 223653
\end{aligned}
\tag{B.23}
$$

Setting equation B.23 to equal zero and solving for `InvokeCount` gives `InvokeCount` = 3.955. This means that energy savings is possible when invoking the service remotely relative to locally if the service is invoked at least 4 times. The cost of invoking the service locally versus remotely and the potential energy savings is shown in Figure 6.16.

B.4.2 Validation of Equations

To validate the equations and conclusions of Section B.4.1, the system is deployed on a network consisting of one Imote2 device and one TelosB device. The same parameters from Section B.4 are kept, specifically `DutyCycle` = 10 and `InvokePeriod` = 1000. Since Servilla concluded that at least 4 invocations must occur before a net savings is achieved when invoking remotely versus locally, `InvokeCount` is varied from 2 through 10. Both local invocations and remote invocations are evaluated. Each configuration and `InvokeCount` combination is evaluated 5 times, enabling the calculation of the average and 95% confidence interval. The results are plotted in Figure 6.16.

The results indicate that the equations derived in Section B.4.1 successfully determine when invoking a service remotely results in energy-savings. Specifically, the measured results also conclude that the remote service must be invoked at least four times for energy savings to occur. The calculated results and measured results do differ slightly in that the point of intersection for the calculated results is 3.95, while the point of intersection for the measured data is 3.45. One possible reason for this discrepancy is due to the limitations of the oscilloscope, which must be configured at a significantly coarser resolution (i.e., $4ms$ versus $0.1ms$ per sample) to capture the entire service invocation process, which may last up to 10 seconds.

References

[1] João Abreu and José Luiz Fiadeiro. A coordination model for service-oriented interactions. In Lea and Zavattaro [95], pages 1–16.

[2] Anurag Acharya, M. Ranganathan, and Joel Saltz. Sumatra: A Language for Resource-aware Mobile Programs. In J. Vitek and C. Tschudin, editors, *Mobile Object Systems: Towards the Programmable Internet*, volume 1222, pages 111–130. Springer-Verlag: Heidelberg, Germany, 1997.

[3] Gustavo Alonso, Fabio Casati, Harumi Kuno, and Vijay Machiraju. *Web Services*. Springer, 2003.

[4] Giuseppe Anastasi, Marco Conti, Mario Di Francesco, and Andrea Passarella. Energy conservation in wireless sensor networks: A survey. *Ad Hoc Netw.*, 7(3):537–568, 2009.

[5] Anupriya Ankolekar, Frank Huch, and Katia P. Sycara. Concurrent semantics for the web services specification language daml-s. In Arbab and Talcott [8], pages 14–21.

[6] Th. Arampatzis, J. Lygeros, and S. Manesis. A survey of applications of wireless sensors and wireless sensor networks. In *Proc. of the 13th Med. Conf. on Control and Automation*, pages 719–724, June 2005.

[7] Farhad Arbab, Tom Chothia, Sun Meng, and Young-Joo Moon. Component connectors with qos guarantees. In Murphy and Vitek [127], pages 286–304.

[8] Farhad Arbab and Carolyn L. Talcott, editors. *Coordination Models and Languages, 5th International Conference, COORDINATION 2002, YORK, UK, April 8-11, 2002, Proceedings*, volume 2315 of *Lecture Notes in Computer Science*. Springer, 2002.

[9] Arch Rock. Arch Rock PhyNet^{TM}. `http://www.archrock.com/product/`.

[10] Edgardo Avilés-López and J. García-Macías. Tinysoa: a service-oriented architecture for wireless sensor networks. *Service Oriented Computing and Applications*, April 2009.

[11] Özalp Babaoglu, Hein Meling, and Alberto Montresor. Anthill: A framework for the development of agent-based peer-to-peer systems. In *ICDCS '02: Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 15, Washington, DC, USA, 2002. IEEE Computer Society.

[12] Amol Bakshi, Viktor K. Prasanna, Jim Reich, and Daniel Larner. The abstract task graph: a methodology for architecture-independent programming of networked sensor systems. In *EESR '05: Proceedings of the 2005 workshop on End-to-end, sense-and-respond systems, applications and services*, pages 19–24, Berkeley, CA, USA, 2005. USENIX Association.

[13] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 112–121, New York, NY, USA, 2006. ACM.

[14] Rahul Balani, Chih-Chieh Han, Ram Kumar Rengaswamy, Ilias Tsigkogiannis, and Mani Srivastava. Multi-level software reconfiguration for sensor networks. In *EMSOFT '06: Proceedings of the 6th ACM & IEEE International conference on Embedded software*, pages 112–121, New York, NY, USA, 2006. ACM.

[15] Mario Baldi and Gian Pietro Picco. Evaluating the tradeoffs of mobile code design paradigms in network management applications. In *ICSE '98: Proceedings of the 20th international conference on Software engineering*, pages 146–155, Washington, DC, USA, 1998. IEEE Computer Society.

[16] Nilanjan Banerjee, Jacob Sorber, Mark D. Corner, Sami Rollins, and Deepak Ganesan. Triage: balancing energy and quality of service in a microserver. In *MobiSys '07: Proceedings of the 5th international conference on Mobile systems, applications and services*, pages 152–164, New York, NY, USA, 2007. ACM.

[17] M. A. Batalin, M. Rahimi, Y.Yu, D.Liu, A.Kansal, G.S. Sukhatme, W.J. Kaiser, M.Hansen, G. J. Pottie, M. Srivastava, and D. Estrin. Towards event-aware adaptive sampling using static and mobile nodes. Technical Report 38, Center for Embedded Networked Sensing, 2004.

[18] J. Baumann, Hohl K. Rothermel, M. Strasser, and W. Theilmann. Mole: A mobile agent system. *Softw. Pract. Exper.*, 32(6):575–603, 2002.

[19] O. Burchan Bayazit, Jyh-Ming Lien, and Nancy M. Amato. Roadmap-based flocking for complex environments. In *Proceedings of the 10th Pacific Conference on Computer Graphics and Applications (PG'02)*, pages 104–121, 2002.

[20] Lorenzo Bettini, Rocco De Nicola, and Michele Loreti. Implementing session centered calculi. In Lea and Zavattaro [95], pages 17–32.

[21] Sangeeta Bhattacharya, Nuzhet Atay, Gazihan Alankus, Chenyang Lu, O. Burchan Bayazit, and Gruia-Catalin Roman. Roadmap query for sensor network assisted navigation in dynamic environments. Technical Report WUCSE-05-41, Washington University in St.Louis, 2005.

[22] Sangeeta Bhattacharya, Nuzhet Atay, Gazihan Alankus, Chenyang Lu, O. Burchan Bayazit, and Gruia-Catalin Roman. Roadmap query for sensor network assisted navigation in dynamic environments. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems (DCOSS)*, pages 17–36, 2006.

[23] Laura Bocchi, Paolo Ciancarini, and Davide Rossi. Transactional aspects in semantic based discovery of services. In Jacquet and Picco [80], pages 283–297.

[24] Laura Bocchi and Roberto Lucchi. Atomic commit and negotiation in service oriented computing. In Ciancarini and Wiklicky [36], pages 16–27.

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

[26] Mario Bravetti and Gianluigi Zavattaro. A theory for strong service compliance. In Murphy and Vitek [127], pages 96–112.

[27] Antonio Brogi, Jean-Marie Jacquet, and Isabelle Linden. On modeling coordination via asynchronous communication and enhanced matching. In Antonio Brogi and Jean-Marie Jacquet, editors, *Electronic Notes in Theoretical Computer Science*, volume 68. Elsevier, 2003.

[28] Roberto Bruni, Ivan Lanese, Hernán C. Melgratti, and Emilio Tuosto. Multiparty sessions in soc. In Lea and Zavattaro [95], pages 67–82.

[29] G. Cabri, L. Leonardi, and F. Zambonelli. MARS: A programmable coordination architecture for mobile agents. *Internet Computing*, 4(4):26–35, 2000.

[30] Alberto Cerpa, Jeremy Elson, Deborah Estrin, Lewis Girod, Michael Hamilton, and Jerry Zhao. Habitat monitoring: application driver for wireless communications technology. *SIGCOMM Comput. Commun. Rev.*, 31(2 supplement):20–41, 2001.

[31] Dipanjan Chakraborty and Harry Chen. Service discovery in the future for mobile commerce. *Crossroads*, 7(2):18–24, 2000.

[32] Stuart Cheshire. DNS-based service discovery. Technical report, Apple Computer, Inc., 2005.

[33] Krishna Chintalapudi, Tat Fu, Jeongyeup Paek, Nupur Kothari, Sumit Rangwala, John Caffrey, Ramesh Govindan, Erik Johnson, and Sami Masri. Monitoring civil structures with a wireless sensor network. *IEEE Internet Computing*, 10(2):26–34, 2006.

[34] Octav Chipara, Christopher Brooks, Sangeeta Bhattacharya, Chenyang Lu, Roger Chamberlain, Gruia-Catalin Roman, and Thomas C. Bailey. Reliable data collection from mobile users for real-time clinical monitoring. Technical Report WUCSE-2008-25, Washington University in St. Louis, December 2008.

[35] Young-Geun Choi, Jeonil Kang, and DaeHun Nyang. Proactive code verification protocol in wireless sensor network. *Lecture Nodes in Computer Science*, 4706(6):1085–1096, 2007.

[36] Paolo Ciancarini and Herbert Wiklicky, editors. *Coordination Models and Languages, 8th International Conference, COORDINATION 2006, Bologna, Italy, June 14-16, 2006, Proceedings*, volume 4038 of *Lecture Notes in Computer Science*. Springer, 2006.

[37] Crossbow Technologies. Imote2 datasheet. `http://tinyurl.com/5jrw85`.

[38] Crossbow Technology. Mica2 wireless measurement system. http://www.xbow.com/Products/productdetails.aspx?sid=174, February 2005.

[39] Crossbow Technology. MicaZ wireless measurement system. http://www.xbow.com/Products/productdetails.aspx?sid=164, February 2005.

[40] Javier Cubo, Gwen Salaün, Javier Cámara, Carlos Canal, and Ernesto Pimentel. Context-based adaptation of component behavioural interfaces. In Murphy and Vitek [127], pages 305–323.

[41] Gianpaolo Cugola, Elisabetta Di Nitto, and Alfonso Fuggetta. The JEDI event-based infrastructure and its application to the development of the OPSS WFMS. *IEEE Transactions on Software Engineering*, 27(9):827–850, September 2001.

[42] Gianpaolo Cugola and Gian Pietro Picco. Peerware: Core middleware support for peer-to-peer and mobile systems. Technical report, Politecnico di Milano, 2001.

[43] David Culler, Deborah Estrin, and Mani Srivastava. Overview of sensor networks. *IEEE Computer*, 37(8):41–49, 2004.

[44] Dave Marshall. Remote procedure calls (rpc). `http://www.cs.cf.ac.uk/Dave/C/node33.html`.

[45] Alexander Davis and Du Zhang. A comparative study of soap and dcom. *J. Syst. Softw.*, 76(2):157–169, 2005.

[46] Jessie Dedecker. Ambient-oriented programming in ambienttalk: combining mobile hardware with simplicity and expressiveness. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 196–197, New York, NY, USA, 2005. ACM.

[47] Adam Dunkels. A low-overhead script language for tiny networked embedded systems. Technical Report T2006:15, Swedish Institute of Computer Science, September 2006.

[48] Adam Dunkels, Bjorn Gronvall, and Thiemo Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *LCN '04: Proceedings of the 29th Annual IEEE International Conference on Local Computer Networks*, pages 455–462, Washington, DC, USA, 2004. IEEE Computer Society.

[49] Elliot Berk. Jlex: A lexical analyzer generator for java. `http://www.cs.princeton.edu/~appel/modern/java/JLex/`.

[50] R. Englemore and T. Morgan. *Blackboard systems.* Addison-Wesley Publishing Company, 1988.

[51] David Flanagan. *JavaScript: The Definitive Guide, 4th Ed.* O'REILLY, Inc., 2001.

[52] Chien-Liang Fok. Agilla Website. http://mobilab.wustl.edu/projects/agilla.

[53] Chien-Liang Fok, Gruia-Catalin Roman, and Chenyang Lu. Mobile agent middleware for sensor networks: an application case study. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 51, Piscataway, NJ, USA, 2005. IEEE Press.

[54] Rodrigo Fonseca, Omprakash Gnawali, Kyle Jamieson, and Philip Levis. Four bit wireless link estimation. In *In Proceedings of the Sixth Workshop on Hot Topics in Networks (HotNets VI)*, 2007.

[55] Davide Frey and Gruia-Catalin Roman. Context-aware publish subscribe in mobile ad hoc networks. In Murphy and Vitek [127], pages 37–55.

[56] David Gay, Philip Levis, Robert von Behren, Matt Welsh, Eric Brewer, and David Culler. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 1–11, New York, NY, USA, 2003. ACM.

[57] D. Gelernter. Generative communication in Linda. *ACM Trans. on Prog. Languages and Systems*, 7(1):80–112, 1985.

[58] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Ctp: Robust and efficient collection through control and data plane integration. Technical Report SING-08-02, Stanford Information Networks Group, February 2008.

[59] Omprakash Gnawali, Rodrigo Fonseca, Kyle Jamieson, David Moss, and Philip Levis. Collection Tree Protocol. In *Proceedings of the 7th ACM Conference on Embedded Networked Sensor Systems (SenSys'09)*, November 2009.

[60] Omprakash Gnawali, Ki-Young Jang, Jeongyeup Paek, Marcos Vieira, Ramesh Govindan, Ben Greenstein, August Joki, Deborah Estrin, and Eddie Kohler. The tenet architecture for tiered sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 153–166, New York, NY, USA, 2006. ACM.

[61] R. Gray. Agent Tcl. *Dr. Dobb's Journal of Software Tools*, 22(3):18–71, 1997.

[62] Ben Greenstein, Eddie Kohler, and Deborah Estrin. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 69–80, New York, NY, USA, 2004. ACM.

[63] Ramakrishna Gummadi, Nupur Kothari, Ramesh Govindan, and Todd Millstein. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles*, pages 1–2, New York, NY, USA, 2005. ACM.

[64] Gregory Hackmann, Octav Chipara, and Chenyang Lu. Robust topology control for indoor wireless sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 57–70, New York, NY, USA, 2008. ACM.

[65] Gregory Hackmann, Chien-Liang Fok, Gruia-Catalin Roman, Chenyang Lu, Christopher Zuver, Kent English, and John Meier. Demo abstract: Agile cargo tracking using mobile agents. In *Proceedings of the 3rd Annual Conference on Embedded Networked Sensor Systems (SenSys'05)*, page 303. ACM, November 2005.

[66] Gregory Hackmann, Fei Sun, Nestor Castaneda, Chenyang Lu, and Shirley Dyke. A holistic approach to decentralized structural damage localization using wireless sensor networks. In *RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium*, pages 35–46, Washington, DC, USA, 2008. IEEE Computer Society.

[67] Brian Hall. *Beej's Guide to Network Programming.* Jorgensen Publishing, January 2009.

[68] Chih-Chieh Han, Ram Kumar, Roy Shea, Eddie Kohler, and Mani Srivastava. A dynamic operating system for sensor nodes. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 163–176, New York, NY, USA, 2005. ACM.

[69] Radu Handorean and Gruia-Catalin Roman. Service provision in ad hoc networks. In Arbab and Talcott [8], pages 207–219.

[70] Tian He, Chengdu Huang, Brian M. Blum, John A. Stankovic, and Tarek F. Abdelzaher. Range-free localization and its impact on large scale sensor networks. *ACM Trans. Embed. Comput. Syst.*, 4(4):877–906, 2005.

[71] Tian He, Sudha Krishnamurthy, Liqian Luo, Ting Yan, Lin Gu, Radu Stoleru, Gang Zhou, Qing Cao, Pascal Vicaire, John A. Stankovic, Tarek F. Abdelzaher, Jonathan Hui, and Bruce Krogh. Vigilnet: An integrated sensor network system for energy-efficient surveillance. *ACM Trans. Sen. Netw.*, 2(1):1–38, 2006.

[72] Tian He, Sudha Krishnamurthy, John A. Stankovic, Tarek Abdelzaher, Liqian Luo, Radu Stoleru, Ting Yan, Lin Gu, Gang Zhou, Jonathan Hui, and Bruce Krogh. Vigilnet:an integrated sensor network system for energy-efficient surveillance. *ACM Transactions on Sensor Networks (under submission)*, 2004.

[73] Jason Hill, Robert Szewczyk, Alec Woo, Seth Hollar, David Culler, and Kristofer Pister. System architecture directions for networked sensors. *SIGPLAN Not.*, 35(11):93–104, 2000.

[74] Timothy W. Hnat, Tamim I. Sookoor, Pieter Hooimeijer, Westley Weimer, and Kamin Whitehouse. Macrolab: a vector-based macroprogramming framework for cyber-physical systems. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 225–238, New York, NY, USA, 2008. ACM.

[75] Tim Tau Hsieh. Using sensor networks for highway and traffic applications. *IEEE Potentials*, 23(2):13–16, Apr-May 2004.

[76] Lingxuan Hu and David Evans. Localization for mobile sensor networks. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 45–57, New York, NY, USA, 2004. ACM.

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

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

[79] Chalermek Intanagonwiwat, Ramesh Govindan, Deborah Estrin, John Heidemann, and Fabio Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Trans. Netw.*, 11(1):2–16, 2003.

[80] Jean-Marie Jacquet and Gian Pietro Picco, editors. *Coordination Models and Languages, 7th International Conference, COORDINATION 2005, Namur, Belgium, April 20-23, 2005, Proceedings*, volume 3454 of *Lecture Notes in Computer Science*. Springer, 2005.

[81] Peter Janacik and Tales Heimfarth. Cross-layer architecture of a distributed os for ad hoc networks. In *ICAS '06: Proceedings of the International Conference on Autonomic and Autonomous Systems*, page 52, Washington, DC, USA, 2006. IEEE Computer Society.

[82] Jaein Jeong. Incremental network programming for wireless sensors. Master's thesis, EECS Department, University of California, Berkeley, 2005.

[83] Dag Johansen, Robbert van Renesse, and Fred B. Schneider. An introduction to the TACOMA distributed system—version 1.0. Technical Report 95-23, University of Tromsø, Tromsø, Norway, June 1995.

[84] Christine Julien and Gruia Catalin Roman. Egocentric context-aware programming in ad hoc mobile environments. *SIGSOFT Softw. Eng. Notes*, 27(6):21–30, 2002.

[85] Chris Karlof, Naveen Sastry, and David Wagner. Tinysec: a link layer security architecture for wireless sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 162–175, New York, NY, USA, 2004. ACM.

[86] James Kempf and Pete St. Pierre. *Service location protocol for enterprise networks: implementing and deploying a dynamic service finder.* John Wiley & Sons, Inc., New York, NY, USA, 1999.

[87] Kevin Vaughan. Wsdlinterpreter. `http://tinyurl.com/67wh2w`.

[88] Young-Jin Kim, Ramesh Govindan, Brad Karp, and Scott Shenker. Geographic routing made practical. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 217–230, Berkeley, CA, USA, 2005. USENIX Association.

[89] Jeffrey King, Raja Bose, Hen-I Yang, Steven Pickles, and Abdelsalam Helal. Atlas: A service-oriented sensor platform. In *Proceedings of the first IEEE International Workshop on Practical Issues in Building Sensor Network Applications (SenseApp 2006)*, pages 630–638, Washington, DC, USA, 2006. IEEE Computer Society.

[90] Joel Koshy and Raju Pandey. VMSTAR: synthesizing scalable runtime environments for sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 243–254, New York, NY, USA, 2005. ACM.

[91] Mark D. Krasniewski, Rajesh Krishna Panta, Saurabh Bagchi, Chin-Lung Yang, and William J. Chappell. Energy-efficient on-demand reprogramming of large-scale sensor networks. *ACM Trans. Sen. Netw.*, 4(1):1–38, 2008.

[92] Ilango Kumaran and S. Ilango Kumaran. *Jini Technology: An Overview*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[93] Manish Kushwaha, Isaac E Amundson, Xenofon Koutsoukos, Sandeep Neema, and Janos Sztipanovits. Oasis: A programming framework for service-oriented sensor networks. In *IEEE/Create-Net COMSWARE 2007*, January 2007.

[94] Danny B. Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Commun. ACM*, 42(3):88–89, 1999.

[95] Doug Lea and Gianluigi Zavattaro, editors. *Coordination Models and Languages, 10th International Conference, COORDINATION 2008, Oslo, Norway, June 4-6, 2008. Proceedings*, volume 5052 of *Lecture Notes in Computer Science*. Springer, 2008.

[96] Philip Levis. Tinyos 2.0 overview. http://www.tinyos.net/tinyos-2.x/doc/html/overview.html.

[97] Philip Levis. The tinyscript language. http://www.cs.berkeley.edu/~pal/mateweb/files/tinyscript-manual.pdf, July 2004.

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

[99] Philip Levis, David Gay, and David Culler. Active sensor networks. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[100] Philip Levis, David Gay, and David Culler. Active sensor networks. In *NSDI'05: Proceedings of the 2nd conference on Symposium on Networked Systems Design & Implementation*, pages 343–356, Berkeley, CA, USA, 2005. USENIX Association.

[101] Philip Levis, Neil Patel, David Culler, and Scott Shenker. Trickle: a self-regulating algorithm for code propagation and maintenance in wireless sensor networks. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 2–2, Berkeley, CA, USA, 2004. USENIX Association.

[102] Lily Li and Kerry Taylor. A framework for semantic sensor network services. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 347–361, Berlin, Heidelberg, 2008. Springer-Verlag.

[103] Tsung-Hsien Lin. http://www.janet.ucla.edu/WINS/.

[104] Tsung-Hsien Lin, Henry Sanchez, William J. Kaiser, and Henry Marcy. Wireless integrated network sensors (wins) for tactical information systems. In *Proc. of the 1998 Government Microcircuit Applications Conference*, 1998.

[105] Hongzhou Liu, Tom Roeder, Kevin Walsh, Rimon Barr, and Emin Gün Sirer. Design and implementation of a single system image operating system for ad hoc networks. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 149–162, New York, NY, USA, 2005. ACM.

[106] Jie Liu and Feng Zhao. Towards semantic services for sensor-rich information systems. In *2nd Int. Conf. on Broadband Networks*, pages 44–51, 2005.

[107] Ting Liu and Margaret Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *PPoPP '03: Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118, New York, NY, USA, 2003. ACM.

[108] Konrad Lorincz, David Malan, Thaddeus R. F. Fulford-Jones, Alan Nawoj, Antony Clavel, Victor Shnayder, Geoff Mainland, Steve Moulton, and Matt Welsh. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing, Special Issue on Pervasive Computing for First Response*, pages 16–23, Oct-Dec 2004.

[109] Dimitrios Lymberopoulos, Bodhi Priyantha, Michel Goraczko, and Feng Zhao. Towards energy efficient design of multi-radio platforms for wireless sensor networks. In *IPSN'08*. IEEE, April 2008.

[110] Dimitrios Lymberopoulos, Nissanka B. Priyantha, and Feng Zhao. mplatform: a reconfigurable architecture and efficient data sharing mechanism for modular sensor nodes. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 128–137, New York, NY, USA, 2007. ACM.

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

[112] Pattie Maes, Robert H. Guttman, and Alexandros G. Moukas. Agents that buy and sell. *Commun. ACM*, 42(3):81–ff., 1999.

[113] Alan Mainwaring, Joseph Polastre, Robert Szewczyk, David Culler, and John Anderson. Wireless sensor networks for habitat monitoring. In *Proc. of the 1st ACM Workshop on Wireless Sensor Networks and Applications*, September 2002.

[114] Miklós Maróti, Branislav Kusy, Gyula Simon, and Ákos Lédeczi. The flooding time synchronization protocol. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 39–49, New York, NY, USA, 2004. ACM.

[115] Pedro José Marrón, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel. Flexcup: A flexible and efficient code update mechanism for sensor networks. In Römer et al. [154], pages 212–227.

[116] Pedro José Marrón, Daniel Minder, Andreas Lachenmann, and Kurt Rothermel. TinyCubus: An adaptive cross-layer framework for sensor networks. *it - Information Technology*, 47(2):87–97, 2005.

[117] David Marsh, Donal O'Kane, and G. M. P. O'Hare. Agents for wireless sensor network power management. In *ICPPW '05: Proceedings of the 2005 International Conference on Parallel Processing Workshops*, pages 413–418, Washington, DC, USA, 2005. IEEE Computer Society.

[118] Manuel Mazzara and Sergio Govoni. A case study of web services orchestration. In Jacquet and Picco [80], pages 1–16.

[119] Elena Meshkova, Janne Riihijarvi, Frank Oldewurtel, Christine Jardak, and Petri Mahonen. Service-oriented design methodology for wireless sensor networks: A view through case studies. *Sensor Networks, Ubiquitous, and Trustworthy Computing, International Conference on*, 0:146–153, 2008.

[120] Leonardo Gaetano Mezzina. How to infer finite session types in a calculus of services and sessions. In Lea and Zavattaro [95], pages 216–231.

[121] Microsoft. Windows communication foundation. http://msdn2.microsoft.com/en-us/library/ms735119.aspx.

[122] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, i. *Inf. Comput.*, 100(1):1–40, 1992.

[123] Robin Milner, Joachim Parrow, and David Walker. A calculus of mobile processes, ii. *Inf. Comput.*, 100(1):41–77, 1992.

[124] David Moss and Philip Levis. BoX-MACs: Exploiting physical and link layer boundaries in low-power networking. Technical Report SING-08-00, Rincon Resarch Corporation and Stanford University, 2008.

[125] René Müller, Gustavo Alonso, and Donald Kossmann. A virtual machine for sensor networks. *SIGOPS Oper. Syst. Rev.*, 41(3):145–158, 2007.

[126] Amy L. Murphy, Gian Pietro Picco, and Gruia-Catalin Roman. Lime: A coordination model and middleware supporting mobility of hosts and agents. *ACM Trans. Softw. Eng. Methodol.*, 15(3):279–328, 2006.

[127] Amy L. Murphy and Jan Vitek, editors. *Coordination Models and Languages, 9th International Conference, COORDINATION 2007, Paphos, Cyprus, June 6-8, 2007, Proceedings*, volume 4467 of *Lecture Notes in Computer Science*. Springer, 2007.

[128] Rohan Murty, Abhimanyu Gosain, Matthew Tierney, Andrew Brody, Amal Fahad, Josh Bers, and Matt Welsh. Citysense: A vision for an urban-scale wireless networking testbed. Technical Report 13-07, Harvard University, 2007.

[129] George C. Necula. Proof-carrying code. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 106–119, New York, NY, USA, 1997. ACM.

[130] Yang Ni, Ulrich Kremer, Adrian Stere, and Liviu Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. *SIGPLAN Not.*, 40(6):249–260, 2005.

[131] Martín López Nores, Jorge García Duque, and José J. Pazos Arias. Managing ad-hoc networks through the formal specification of service requirements. In Ciancarini and Wiklicky [36], pages 164–178.

[132] Angel Núñez and Jacques Noyé. An event-based coordination model for context-aware applications. In Lea and Zavattaro [95], pages 232–248.

[133] Object Management Group. Corba basics. http://www.omg.org/gettingstarted/corbafaq.htm.

[134] OSGi. Open source gateway initiative. `http://www.osgi.org`.

[135] OSGi Allance. Open service gateway initiative. http://www.osgi.org.

[136] Michael P. Papazoglou, Paolo Traverso, Schahram Dustdar, and Frank Leymann. Service-oriented computing: State of the art and research challenges. *Computer*, 40(11):38–45, 2007.

[137] Mike P. Papazoglou. Service -oriented computing: Concepts, characteristics and directions. *Web Information Systems Engineering, International Conference on*, 0:3, 2003.

[138] Animesh Pathak and Viktor K. Prasanna. Energy-efficient task mapping for data-driven sensor network macroprogramming. In *DCOSS '08: Proceedings of the 4th IEEE international conference on Distributed Computing in Sensor Systems*, pages 516–524, Berlin, Heidelberg, 2008. Springer-Verlag.

[139] P.E.Clements, Todd Papaioannou, and John Edwards. Aglets: Enabling the virtual enterprise. In *Proc. of the Int. Conf. on Managing Enterprises - Stakeholders, Engineering, Logistics and Achievement*, 1997.

[140] H. Peine and T. Stolpmann. The architecture of the Ara platform for mobile agents. In Radu Popescu-Zeletin and Kurt Rothermel, editors, *First International Workshop on Mobile Agents MA'97*, volume 1219 of *Lecture Notes in Computer Science*, pages 50–61, Berlin, Germany, April 1997. Springer Verlag.

[141] Adrian Perrig, John Stankovic, and David Wagner. Security in wireless sensor networks. *Commun. ACM*, 47(6):53–57, 2004.

[142] Gian Pietro Picco. ucode: A lightweight and flexible mobile code toolkit. In Kurt Rothermel and Fritz Hohl, editors, *Proceedings of the 2nd International Workshop on Mobile Agents*, Lecture Notes in Computer Science, pages 160–171, Berlin, Germany, 1998. Springer-Verlag.

[143] Eric Platon and Yuichi Sei. Security engineering in wireless sensor networks. *Progress in Informatics*, 5(3):49–64, 2008.

[144] Joseph Polastre, Robert Szewczyk, and David Culler. Telos: enabling ultra-low power wireless research. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks*, page 48, Piscataway, NJ, USA, 2005. IEEE Press.

[145] Lucia Del Prete and Licia Capra. Reliable discovery and selection of composite services in mobile environments. In *EDOC '08: Proceedings of the 2008 12th International IEEE Enterprise Distributed Object Computing Conference*, pages 171–180, Washington, DC, USA, 2008. IEEE Computer Society.

[146] Jaco M. Prinsloo, Christian L. Schulz, Derrick G. Kourie, W. H. Morkel Theunissen, Tinus Strauss, Roelf Van Den Heever, and Sybrand Grobbelaar. A service oriented architecture for wireless sensor and actor network applications. In *SAICSIT '06: Proceedings of the 2006 annual research conference of the South African institute of computer scientists and information technologists on IT research in developing countries*, pages 145–154, , Republic of South Africa, 2006. South African Institute for Computer Scientists and Information Technologists.

[147] Nissanka B. Priyantha, Aman Kansal, Michel Goraczko, and Feng Zhao. Tiny web services: design and implementation of interoperable and evolvable sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems*, pages 253–266, New York, NY, USA, 2008. ACM.

[148] Hairong Qi, S. S. Iyengar, and Krishnendu Chakrabarty. Multiresolution data integration using mobile agents in distributed sensor networks. *IEEE Trans. on Systems, Man, and Cybernetics – Part C*, 31(3):383–391, August 2001.

[149] Hairong Qi, Xiaoling Wang, S. Sitharama Iyengar, and Krishnendu Chakrabarty. Multisensor data fusion in distributed sensor networks using mobile agents. In *In Proceedings of 5th International Conference on Information Fusion*, pages 11–16, August 2001.

[150] Hairong Qi, Yingyue Xu, and Xiaoling Wang. Mobile-agent-based collaborative signal and information processing in sensor networks. In *Proc. of the IEEE*, volume 91, pages 1172–1183. IEEE, August 2003.

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

[152] Gruia-Catalin Roman, Qingfeng Huang, and Ali Hazemi. Consistent group membership in ad hoc networks. In *ICSE '01: Proceedings of the 23rd International Conference on Software Engineering*, pages 381–388, Washington, DC, USA, 2001. IEEE Computer Society.

[153] Gruia-Catalin Roman, Peter J. McCann, and Jerome Y. Plun. Mobile UNITY: reasoning and specification in mobile computing. *ACM Transactions on Software Engineering and Methodology*, 6(3):250–282, 1997.

[154] Kay Römer, Holger Karl, and Friedemann Mattern, editors. *Wireless Sensor Networks, Third European Workshop, EWSN 2006, Zurich, Switzerland, February 13-15, 2006, Proceedings*, volume 3868 of *Lecture Notes in Computer Science*. Springer, 2006.

[155] Masoomeh Rudafshani and Suprakash Datta. Localization in wireless sensor networks. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks*, pages 51–60, New York, NY, USA, 2007. ACM.

[156] Silvia Santini and Kay Rmer. An adaptive strategy for quality-based data reduction in wireless sensor networks. In *Proceedings of the 3rd International Conference on Networked Sensing Systems (INSS 2006)*, pages 29–36, Chicago, IL, USA, June 2006. TRF.

[157] Andreas Scholz, Christian Buckl, Stephan Sommer, Alfons Kemper, Alois Knoll, Jrg Heuer, and Anton Schmitt. esoa - service oriented architectures adapted for embedded networks. In *IDIN 2009: 7th International Conference on Industrial Informatics*, pages 599–605, Los Alamitos, CA, USA, June 2009. IEEE.

[158] Scott Hudson. CUP LALR Parser Generator for Java. `http://www2.cs.tum.edu/projects/cup/`.

[159] B. Sklar. Rayleigh fading channels in mobile digital communication systems .i. characterization. *Communications Magazine, IEEE*, 35(7):90–100, 1997.

[160] Stephan Sommer, Christian Buckl, and Alois Knoll. Developing service oriented sensor/actuator networks using a tailored middleware. *Information Technology: New Generations, Third International Conference on*, 0:1036–1041, 2009.

[161] SourceForge. `http://platformx.sourceforge.net/`.

[162] William Stallings. *Operating Systems 4th Ed.* Pretence Hall, New Jersey, 4 edition, 2001.

[163] Thanos Stathopoulos, John Heidemann, and Deborah Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS-TR-30, UCLA CENS, 2003.

[164] Radu Stoleru, Tian He, John A. Stankovic, and David Luebke. A high-accuracy, low-cost localization system for wireless sensor networks. In *SenSys '05: Proceedings of the 3rd international conference on Embedded networked sensor systems*, pages 13–26, New York, NY, USA, 2005. ACM.

[165] Streeline. Parking management. `http://www.streetlinenetworks.com`.

[166] Lang Tong, Qing Zhao, and Srihari Adireddy. Sensor networks with mobile agents. In *in Proc. 2003 Military Communications Intl Symp*, pages 688–693, 2003.

[167] Yu-Chee Tseng, Sheng-Po Kuo, Hung-Wei Lee, and Chi-Fu Huang. Location Tracking in a Wireless Sensor Network by Mobile Agents and Its Data Fusion Strategies. *The Computer Journal*, 47(4):448–460, 2004.

[168] Yu-Chee Tseng, Sheng-Po Kuo, Wung-Wei Lee, and Chi-Fu Huang. Location tracking in a wireless sensor network by mobile agents and its data fusion strategies. *The Computer Journal*, 47(4):448–460, 2004.

[169] Richard Tynan, Antonio G. Ruzzelli, and O'Hare G. M. P. A methodology for the development of multi-agent systems on wireless sensor networks. *In Proc. the 17th International Conference on Software Engineering and Knowledge Engineering*, July 2005.

[170] Tyndall National Institute. The 25mm cube module. http://www.tyndall.ie/research/mai-group/25cube_mai.html, September 2005.

[171] W3C. Web services description language (wsdl). `http://www.w3.org/TR/wsdl`.

[172] W3C Architecture Domain. Xml schema. `http://www.w3.org/XML/Schema`.

[173] Chieh-Yih Wan, Andrew T. Campbell, and Lakshman Krishnamurthy. PSFQ: a reliable transport protocol for wireless sensor networks. In *WSNA '02: Proceedings of the 1st ACM international workshop on Wireless sensor networks and applications*, pages 1–11, New York, NY, USA, 2002. ACM.

[174] Brett Warneke, Matt Last, Brian Liebowitz, and Kristofer S. J. Pister. Smart dust: Communicating with a cubic-millimeter computer. *Computer*, 34(1):44–51, 2001.

[175] Washington University in St. Louis. WSN testbed. `http://tinyurl.com/yjuctmb`.

[176] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

[177] Matt Welsh and Geoff Mainland. Programming sensor networks using abstract regions. In *NSDI'04: Proceedings of the 1st conference on Symposium on Networked Systems Design and Implementation*, pages 3–3, Berkeley, CA, USA, 2004. USENIX Association.

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

[179] Winamp. http://www.winamp.com/.

[180] Alec Woo, Terence Tong, and David Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 14–27, New York, NY, USA, 2003. ACM.

[181] Michael Wooldridge and Nick Jennings. Intelligent agents: Theory and practice. *IEEE Trans. on Knowledge Engineering Review*, 10(2):115–152, 1995.

[182] Qishi Wu, Nageswara Rao, Jacob Barhen, S. Sitharama Iyengar, Vijay Vaishnavi, Hairong Qi, and Krishnendu Chakrabarty. On computing mobile agent routes for data fusion in distributed sensor networks. *IEEE Trans. on Knowledge and Data Engineering*, 6(16):740–753, June 2004.

[183] Ning Xu, Sumit Rangwala, Krishna Kant Chintalapudi, Deepak Ganesan, Alan Broad, Ramesh Govindan, and Deborah Estrin. A wireless sensor network for structural monitoring. In *Proc. of the 2nd Int. Conf. on Embedded Networked Sensor Systems (SenSys '04)*, pages 13–24. ACM Press, 2004.

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

[185] Yang Yu, Loren J. Rittle, Vartika Bhandari, and Jason B. LeBrun. Supporting concurrent applications in wireless sensor networks. In *SenSys '06: Proceedings of the 4th international conference on Embedded networked sensor systems*, pages 139–152, New York, NY, USA, 2006. ACM.

[186] Jerry Zhao and Ramesh Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 1–13, New York, NY, USA, 2003. ACM.

Vita

Chien-Liang Fok

Date of Birth March 16, 1980

Place of Birth New York, New York

Degrees B.S. Magna Cum Laude, Computer Science and Engineering, May 2002
Ph.D. Computer Science, December 2009

Publications Fok, C., Roman, G., and Lu, C. Agilla: A Mobile Agent Middleware for Self-Adaptive Wireless Sensor Networks. In ACM Trans. Auton. Adapt. Syst. Special Issue on Self-Adaptive and Self-Organizing Wireless Networking Systems. 4, 3 (Jul. 2009), 1-26.

Bhattacharya, S., Fok, C., Lu, C., and Roman, G. MLDS: A Flexible Location Directory Service for Tiered Sensor Networks. In Computer Communications. 31, 6 (Apr. 2008), 1160-1172.

Fok, C.-L., Roman, G.-C., and Lu, C. Enhanced Coordination in Sensor Networks through Flexible Service Provisioning. In Proceedings of the 11th International Conference on Coordination Models and Languages (Coordination09), June 2009. Note: Invited to a special issue of Science of Computer Programming on the Best Papers from Coordination'09.

Bhattacharya, S., Fok, C.-L., Lu, C., and Roman, G.-C. Design and Implementation of a Flexible Location Directory Service for Tiered Sensor Networks. In Proceedings of the 3rd International Conference on Distributed Computing in Sensor Systems (DCOSS'07), June 2007.

Hackmann, G., Fok, C.-L., Roman, G.-C., and Lu, C. Agimone: Middleware Support for Seamless Integration of Sensor and IP Networks. In Proceedings of the Second International Conference on Distributed Computing in Sensor Systems (DCOSS'06), June 2006.

Massaguer, D., Fok, C.-L., Venkatasubramanian, N., Roman, G.-C., and Lu. Exploring Sensor Networks using Mobile Agents. In Proceedings of the Fifth International Joint Conference on Autonomous Agents and Multiagent Systems (AAMAS06), May 2006.

Fok, C.-L., Roman, G.-C., and Lu, C. Mobile Agent Middleware for Sensor Networks: An Application Case Study. In Proceedings of the 4th international Symposium on Information Processing in Sensor Networks (IPSN05), April 2005. Google Citation count: 116, Acceptance Ratio: 20.6

Fok, C.-L., Roman, G.-C., and Lu, C. Rapid Development and Flexible Deployment of Adaptive Wireless Sensor Network Applications. In Proceedings of the 25th IEEE international Conference on Distributed Computing Systems (ICDCS05), June 2005. Note: One of 5 papers nominated for Best Paper Award (543 papers submitted). Google Citation Count: 191, Acceptance Ratio: 13.8

Lu, C., Xing, G., Chipara, O., Fok, C.-L., and Bhattacharya, S. A Spatiotemporal Query Service for Mobile Users in Sensor Networks. In Proceedings of the 25th IEEE international Conference on Distributed Computing Systems (ICDCS05), June 2005.

E.H. Clayton, B.H. Koh, G. Xing, C.-L. Fok, S.J. Dyke and C. Lu. Damage Detection and Correlation Based Localization Using Wireless Mote Sensors. In Proceedings of the 13th IEEE Mediterranean Conference on Control and Automation (MED05), June 2005.

Fok, C.-L., Roman, G.-C, and Hackmann, G., A Lightweight Coordination Middleware for Mobile Computing. In Proceedings of the 6th International Conference on Coordination Models and Languages (Coordination04), February 2004.

Fok, C.-L., Roman, G.-C., Lu, C. 2006. Software Support for Application Development in Wireless Sensor Network. A. Corradi and P. Bellavista (editors), Handbook of Mobile Middleware, CRC Press, September 2006.

Fok, C., Roman, G., and Lu, C. 2007. Towards a Flexible Global Sensing Infrastructure. In SIGBED Rev. 4, 3 (Jul. 2007), 1-6.

December 2009