**Application Specific Virtual Machines:
Operating System Support for User-Level Sensornet Programming**

by

Philip Alexander Levis

B.S. (Brown University) 1999

A dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy
in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA, BERKELEY

Committee in charge:

Professor David Culler, Chair
Professor Eric Brewer
Professor Scott Shenker
Professor Paul Wright

Fall 2005

The dissertation of Philip Alexander Levis is approved.

Chair Date

Date

Date

Date

University of California, Berkeley

Fall 2005

Application Specific Virtual Machines:

Operating System Support for User-Level Sensornet Programming

Copyright © 2005

by

Philip Alexander Levis

Abstract

Application Specific Virtual Machines:

Operating System Support for User-Level Sensornet Programming

by

Philip Alexander Levis

Doctor of Philosophy in Computer Science

University of California, Berkeley

Professor David Culler, Chair

We propose using application specific virtual machines (ASVMs) to reprogram deployed wireless sensor networks. ASVMs provide a way for a user to define an application-specific boundary between virtual code and the VM engine. This allows programs to be very concise (tens to hundreds of bytes), making program installation fast and inexpensive. Additionally, concise programs interpret few instructions, imposing very little interpretation overhead. We evaluate ASVMs against current proposals for network programming runtimes and show that ASVMs are more energy efficient by as much as 20%. We also evaluate ASVMs against hand built TinyOS applications and show that while interpretation imposes a significant execution overhead, the low duty cycles of realistic applications make the actual cost effectively unmeasurable.

____

Professor David Culler
Dissertation Committee Chair

Contents

Contents i

List of Figures vii

List of Tables xi

Acknowledgements xii

1 Introduction 1

1.1 Sensor Networks 2
1.2 Problem Statement 5
1.3 Thesis 6
1.4 Maté Overview 7
1.5 Contributions 9
1.6 Assumptions 9
1.7 Summary and Outline 10

2 Background 12

2.1 Application Domains 12
2.1.1 Periodic Sampling 13
2.1.2 Phenomenon Detection 14
2.1.3 Fire Rescue 16
2.1.4 Application Analysis 17
2.2 Hardware Platforms 18
2.2.1 An Example Platform: mica2 20

2.2.2 Microcontroller . . . 20
2.2.3 Radio . . . 22
2.2.4 An Example Platform: Telos revB . . . 23
2.2.5 Hardware Trends . . . 24
2.3 TinyOS . . . 25
2.3.1 Components and Interfaces . . . 26
2.3.2 Parameterized Interfaces . . . 29
2.3.3 Concurrency Model . . . 29
2.3.4 Networking . . . 31
2.3.5 TinyOS Analysis . . . 32
2.4 Architectural Requirements . . . 33
2.5 A Flexible Network Encoding . . . 34

3 Three Program Layers 36
3.1 The Three Layer Model . . . 36
3.1.1 One layer: TinyOS . . . 38
3.1.2 Two layers: SNACK and Regions . . . 38
3.1.3 Two layers: SOS . . . 39
3.1.4 Two layers: Sensorware and TinyDB . . . 40
3.1.5 Three layers: Java and CLR . . . 40
3.1.6 Analysis . . . 41
3.2 Decomposing the Three Layers . . . 42
3.2.1 User Language . . . 43
3.2.2 Network Encoding . . . 43
3.2.3 Node Runtime . . . 44
3.3 A Performance Model . . . 45
3.3.1 Propagation Energy . . . 45
3.3.2 Execution Energy . . . 46
3.3.3 Energy Analysis . . . 47
3.4 Defining the Network Encoding and Runtime . . . 48
3.4.1 The Network Encoding . . . 48
3.4.2 The Underlying Runtime . . . 49

3.4.3 Runtime Transformation . . . 50
3.4.4 Runtime Execution . . . 50
3.4.5 Runtime Propagation . . . 50
3.4.6 Performance Model . . . 51
3.5 Maté Bytecodes As a Network Encoding . . . 51
3.6 Related Work . . . 52

4 The Maté Architecture 55
4.1 Application Specific Virtual Machine Architecture . . . 57
4.2 ASVM Template . . . 57
4.2.1 Execution Model . . . 57
4.2.2 Data Model . . . 58
4.2.3 Scheduler . . . 59
4.2.4 Concurrency Manager . . . 61
4.2.5 Capsule Store . . . 62
4.2.6 Error Handling . . . 62
4.3 ASVM Extensions . . . 62
4.3.1 Operations . . . 63
4.3.2 Handlers . . . 64
4.4 Concurrency Management . . . 64
4.4.1 Implicit Synchronization . . . 64
4.4.2 Synchronization Model . . . 65
4.4.3 Synchronization Algorithm . . . 66
4.4.4 Deadlock-free . . . 67
4.4.5 Implementation . . . 68
4.5 Program Encoding . . . 69
4.5.1 Bytecodes . . . 70
4.5.2 Compilation and Assembly . . . 72
4.6 Building an ASVM . . . 72
4.6.1 Languages . . . 74
4.6.2 Libraries . . . 74
4.6.3 Toolchain . . . 77

4.6.4 Example Scripts: Bombilla . . . 77
4.7 Evaluation . . . 80
4.7.1 Efficiency . . . 80
4.7.2 Conciseness . . . 83
4.7.3 Whole System Evaluation . . . 87
4.7.4 Cost/Benefit Tradeoff . . . 93
4.7.5 Analysis . . . 94
4.8 Related Work . . . 95

5 Code Propagation **98**
5.1 Trickle Algorithm . . . 100
5.1.1 Overview . . . 101
5.2 Methodology . . . 102
5.2.1 Abstract Simulation . . . 102
5.2.2 TOSSIM . . . 102
5.2.3 TinyOS motes . . . 104
5.3 Maintenance . . . 104
5.3.1 Maintenance with Loss . . . 106
5.3.2 Maintenance without Synchronization . . . 108
5.3.3 Maintenance in a Multi-hop Network . . . 111
5.3.4 Load Distribution . . . 114
5.3.5 Empirical Study . . . 116
5.4 Propagation . . . 117
5.4.1 Propagation Protocol . . . 118
5.4.2 Simulation . . . 120
5.4.3 Empirical Study . . . 122
5.4.4 State . . . 124
5.5 ASVM Protocol . . . 124
5.5.1 The Three Trickles . . . 126
5.6 Protocol Evaluation . . . 126
5.6.1 Parameters . . . 127
5.6.2 Methodology . . . 127

5.6.3 Propagation Rate 128
5.6.4 Propagation Cost 131
5.6.5 Analysis 131
5.7 Security 131
5.8 Related Work 132

6 Macrosystem Evaluation: Use Cases 135
6.1 SmokeVM: Configuration and Management 135
6.1.1 Fire Rescue Application 136
6.1.2 SmokeVM 139
6.1.3 SmokeVM UI 141
6.1.4 Analysis 141
6.2 SaviaVM: Redwood Data Collection 142
6.2.1 SaviaVM 144
6.3 Analysis 145
6.4 Related Work 146

7 Future Work and Discussion 148
7.1 Operating Systems 149
7.1.1 Virtualization 149
7.1.2 User land 150
7.1.3 Customizability 151
7.1.4 How Mote Systems Differ 152
7.2 Network Protocols 153
7.2.1 Metrics 154
7.2.2 Communication Patterns and Administration 154
7.2.3 Network State and Administration 155
7.2.4 Design Considerations for Mote Protocols 156
7.3 Programming Models 157
7.3.1 Parallel Processing 157
7.3.2 Declarative Languages 159
7.3.3 Active Networks 159

7.3.4 Analysis 160
7.4 Discussion 161
7.5 Future Work 162
7.5.1 Execution and Data Model 162
7.5.2 Error Reporting 163
7.5.3 Heterogeneity 163
7.5.4 Propagation 164
7.5.5 ASVM Synthesis 164

8 Conclusion **165**

9 Appendices **167**
9.A SmokeVM and SaviaVM 167
9.B TinyScript 169
9.B.1 TinyScript Grammar 169
9.B.2 TinyScript Primitives 173

Bibliography **175**

List of Figures

1.1 The three program layers. 7

2.1 Profile of the mica2 platform. The table on the left shows the complete breakdown of the subcomponents, their costs, and performance. The chart at the top right shows the relative power draws of the subcomponents when in common operating states. The chart at the bottom right shows the power draw breakdown of a common operating scenario: being fully awake and receiving messages. These numbers represent the costs for the entire platform; for some subsystems, such as the MCU, this includes additional resources beyond a single chip, such as an external oscillator. 19

2.2 Profile of the Telos revB platform. The table on the left shows the complete breakdown of the subcomponents, their costs, and performance. The chart at the top right shows the relative power draws of the subcomponents when in common operating states. The chart at the bottom right shows the power draw breakdown of a common operating scenario, being awake and receiving messages. These numbers represent the costs for the entire platform; for some subsystems, such as the MCU, this includes additional resources beyond a single chip. 23

2.3 TinyOS' BareSendMsg interface, which presents the abstraction of sending a fixed sized message buffer (a TOS_MsgPtr). The nesC interface specifies two call directions. A provider of the interface implements the send command and signals sendDone to the caller; a user of the interface can call the send command, and implements the sendDone handler. 25

2.4 A sample pair of modules, App and Service, which use and provide the BareSendMsg interface of Figure 2.3. As both components are modules, they have implementation code, unlike configurations, which are compositions of components. 27

2.5 nesC examples. 28

3.1 The three program layers. 37

4.1 The ASVM architecture. 56

4.2 The nesC Bytecode interface, which all operations provide. 59

4.3 The OPdivM module implements the divide instruction. The corresponding OPdiv configuration presents a self-contained operation by wiring OPdivM to other components, such as the operand stack ADT (MStacks). The ASVM template component MateEngineM uses a parameterized MateBytecode interface, which represents the ASVM instruction set (the parameter is an opcode value). Wiring OPdiv to MateEngineM includes it in the ASVM instruction set: OP_DIV is the opcode value. MateEngineM runs code by fetching the next opcode from the current handler and dispatches on its MateBytecode interface to execute the corresponding instruction. 60

4.4 Sample Program 1 67

4.5 The BytecodeLock interface. 68

4.6 The ASVM capsule format. 69

4.7 Loading and Storing Shared Variables: Two Maté Primitives Sharing a Single Implementation 71

4.8 Minimal description file for the RegionsVM. Figure 4.14 contains scripts for this ASVM. . . 73

4.9 A simple data collection query in motlle: return node id, parent in routing tree and temperature every 50s. 75

4.10 The TinyScript description file. 76

4.11 The mica sensor board library file. 77

4.12 Bombilla description file. 78

4.13 Bombilla script that loops 500 times, filling a buffer with light readings and sorting the buffer. The program is 32 bytes long. 79

4.14 Regions Pseudocode and Corresponding TinyScript. The pseudocode is from "Programming Sensor Networks Using Abstract Regions." The TinyScript program on the right compiles to 71 bytes of binary code. 85

4.15 An exponentially decaying average operator for TinySQL, in motlle. 88

4.16 Tree topology used in QueryVM/TinyDB experiments. The square node is the tree root. . . . 89

4.17 Power consumption of TinyDB, QueryVM, and nesC implementations. Synch is the nesC implementation when nodes start at the same time. Stagger is when the nodes start times are staggered. 91

4.18 Average power draw measurements in a two node network. For the Conditional query, the monitored node has parent = 0, therefore it sends no packets. The error bars are the standard deviation of the per-interval samples. 92

5.1 TOSSIM packet loss rates over distance 103

5.2 The TinyOS mica2 104

5.3 Trickle maintenance with a k of 1. Dark boxes are transmissions, gray boxes are suppressed transmissions, and dotted lines are heard transmissions. Solid lines mark interval boundaries. Both I_1 and I_2 are of length τ. 105

5.4 Number of transmissions as density increases for different packet loss rates. 107

5.5 The short listen problem for motes A, B, C, and D. Dark bars represent transmissions, light bars suppressed transmissions, and dashed lines are receptions. Tick marks indicate interval boundaries. Mote B transmits in all three intervals. 108

5.6 The short listen problem's effect on scalability, $k = 1$. Without synchronization, Trickle scales with $O(\sqrt{n})$. A listening period restores this to asymptotically bounded by a constant. 109

5.7 Trickle maintenance with a k of 1 and a listen-only period. Dark boxes are transmissions, gray boxes are suppressed transmissions, and dotted lines are heard transmissions. 110

5.8 Simulated trickle scalability for a multi-hop network with increasing density. Motes were uniformly distributed in a 50'x50' square area. 111

5.9 The effect of proximity on the hidden terminal problem. When C is within range of both A and B, CSMA will prevent C from interfering with transmissions between A and B. But when C is in range of A but not B, B might start transmitting without knowing that C is already transmitting, corrupting B's transmission. Note that when A and B are farther apart, the region where C might cause this "hidden terminal" problem is larger. 112

5.10 Communication topography of a simulated 400 mote network in a 20x20 grid with 5 foot spacing (95'x95'), running for twenty minutes with a τ of one minute. The x and y axes represent space, with motes being at line intersections. Color denotes the number of transmissions or receptions at a given mote. 115

5.11 Empirical and simulated scalability over network density. The simulated data is the same as Figure 5.8. 116

5.12 Trickle pseudocode. 117

5.13 Simulated time to code propagation topography in seconds. The hop count values in each legend are the expected number of transmissions necessary to get from corner to corner, considering loss. 119

5.14 Simulated Code Propagation Rate for Different τ_hs. 120

5.15 Empirical testbed layout. 120

5.16 Empirical network propagation time. The graphs on the left show the time to complete reprogramming for 40 experiments, sorted with increasing time. The graphs on the right show the distribution of individual mote reprogramming times for all of the experiments. . . 121

5.17 Maté capsule propagation state machine. 125

5.18 Layout of Soda Hall testbed. The code propagation experiments in Figure 5.19 injected code at mote 473-3 (the bottom left corner). 127

5.19 Reception times for capsules of varying size. Each plot shows the distribution of reception times for the 70-node Soda Hall network over 100 experiments. 129

5.20 Plots of data in Table 5.1. Each plot shows how the packet counts change as capsule size increases. The general trend is a linear increase. The 2 chunk capsule experiments show an aberration in this trend, with higher counts than the 3 chunk capsule. This behavior can be somewhat explained by Figure 5.20(c), which shows the degree (number of receptions per transmission) of each trickle. Some sort of network dynamic or transient interference source possibly changed the network dynamics: chunk transmissions showed a significantly lower degree than other experiments. 130

6.1 SmokeVM physical topology examples. The left of each figure is the physical layout in a building. The white areas represent hallways, while the dark areas represent rooms. The right of each figure is a visualization of each node's physical neighbor table. In Figures 6.1(b), 6.1(c), and 6.1(d), the tables represent up-down rather than left-right for nodes C and D. 138

6.2 The SmokeVM user interface. 140

6.3 A sap flow sensor for redwood trees. The heater in the middle emits a heat pulse. Comparing when the upper and lower sensors observe the pulse provides an estimate of the sap flow rate. In this case, sap is flowing upward. 142

6.4 SaviaVM record format. The status bits include information such as whether the node thinks its global clock is synchronized. 145

List of Tables

2.1 Microcontrollers used in mote platforms. Left to right, they progress in time from first to most recently used. The power values represent maximum draw and assume a 3V supply voltage and operation at 25° C. 21

2.2 Radio chips used in mote platforms. Left to right, they progress in time from earlier to more recently used. The power and energy values assume a 3V supply at 25° C. 22

4.1 Cycle counts (on a mica class mote) for synchronization operations. The costs do not include the overhead of a function call (8 cycles). Depending on what and when the compiler inlines, this overhead may or may not be present. 81

4.2 Cycle counts (on a mica class mote) for common safety checks. The costs do not include the overhead of a function call (8 cycles). Depending on what and when the compiler inlines, this overhead may or may not be present. 82

4.3 Space utilization of native and RegionsVM regions implementations (bytes). 84

4.4 The three queries used to QueryVM versus TinyDB. TinyDB does not directly support time-based aggregates such as `expdecay`, so in TinyDB we omit the aggregate. 86

4.5 Query size in TinyDB and QueryVM. The queries refer to those in Table 4.4. 86

4.6 Query size, power consumption and yield in TinyDB and QueryVM. Yield is the percentage of expected results received. 89

4.7 Execution time of three scripts, in milliseconds. None is the version that did not sort, operation is the version that used an operation, while script is the version that sorted in script code. 93

5.1 Per-node packet transmission and reception counts to reprogram the Soda Hall network for varying capsule sizes. These values represent the averages across 100 experiments on a 73 node mica2dot testbed. 130

6.1 SmokeVM application domain functions. 139

Acknowledgements

The journey through graduate school is ultimately a solitary one — a dissertation bears only one name — but such a simplification conceals all of the people who helped you, guided you, and taught you along the way. It is they who deserve the most recognition and have accomplished the most.

My thanks to and appreciation of David Culler have no bounds. His ability to straddle technological impact as well as intellectual investigation and curiosity showed me that one can be an academic without sequestering in an ivory tower. His cryptic turns of phrase — his “Cullerisms” — were the spark for many of my research ideas and directions.

Eric Brewer and I collaborated on some of my early work, but I am most grateful to him for his advice and guidance. While few in number, his recommendations were always greatly appreciated and tremendously beneficial. I hope some day to have his prescience and vision.

Scott Shenker is a marvel; how a single person can be involved in so many efforts and have so many ideas in his head at once is incomprehensible to me. His humility and respect for others makes him a wonderful role model; I only wish I were a better follower.

Together, the Berkeley faculty showed me that the hope and dream of every graduate is not just a hope and dream: there exist departments whose faculty are foremost concerned with your well-being and accomplishment, and always have your best interests at heart. You can put years of your life safely in their hands, because they will treat your investment well.

David Gay was my compatriot in crime for a great deal of research. He reminded me that convenience and deadlines cannot be a reason for sacrificing quality, and that dedication does not mean sacrificing your life outside of work. Our daily conversations, knock-knocks, and collaboration underlie much of TinyOS and the ideas in this dissertation.

I never would have reached Berkeley were it not for Dirk Grunwald. He helped me in the first few steps of my graduate career, and instilled in me the principle of reading heavily and widely: the best way to build on prior work is to know it well.

My time at Brown lay the groundwork for much of what I have to appreciate in my life today. Its computer science department instilled important principles that have guided all my work since, and the friends I made there have stayed and grown. I'd like to thank Leslie Kaelbling, Philip Klein, Tom Doeppner, Tom Dean, Gideon Mann, Josh Birk, Sarah Cunningham, Sila Ponrartana, Scott Williamson, Anandamayi Arnold, Eric DeRiel, and many others.

Kathryn Williamson has been amazingly patient and supportive, providing a much needed counterpoint to my academic career. I do not think we could have foreseen how auspicious our opportunities in the Bay Area were going to be: together we have been able to achieve more than what our wildest hopes held in June of 2001.

Chapter 1

Introduction

Moore's Law states that the number of transistors in an integrated circuit increases exponentially over time, doubling every 18 months. Since its first mention in 1965, Moore's Law has taken on a much broader scope and significance. Almost all computing resources increase exponentially in size over time, including disk capacity, memory, and network bandwidth. The increase in resources is not accompanied by a corresponding increase in size: a 200GB hard drive in 2005 is the same size as a 200MB hard drive in 1998.

On one hand, Moore's law means that the resources which can fit in a form factor will grow over time. On the other, it also means that existing resources can be made smaller. In the first case, the exponential growth of resources has led existing classes of computing devices to grow more powerful. In the second case, shrinking size has led to the appearance of new, smaller device classes. Minicomputers emerged in the 1970s. Desktops became feasible in the 1980s. The 1990s saw laptops become commonplace, and in the first few years of the 21st century, cellphones are ubiquitous.

A new class of computing devices is on the horizon: embedded wireless sensor networks, collections of small devices with integrated computing, sensing, and networking. Their small size means they can be unobtrusively deployed in a wide range of environments, collecting data at a fidelity and scale that was until now impossible.

1.1 Sensor Networks

The ability to embed sensors in large and uncontrolled environments opens up tremendous possibilities for a wide range of disciplines. Biologists at UC Berkeley have deployed wireless sensor networks in redwood trees to gather real-time data, measuring how a redwood microclimates vary over space and time, as well as how they control and influence that microclimate [23]. Civil engineers have deployed networks on San Francisco's Golden Gate Bridge, to measure how winds and other meteorological conditions affect the bridge [62]. Fire departments are exploring how wireless sensor networks can help rescue trapped people or aid in evacuation [9].

Furthermore, wireless sensor networks have greater potential than simple data collection. When connected to an actuator, a sensor node can actively control its environment. Instead of a single thermostat per floor, homes can have wireless sensor networks that monitor and control home climate on a per room basis or less. In precision agriculture, sensor nodes can measure soil moisture and control irrigation at the granularity of tens of square meters, rather than acres [27]. In fire rescue, a sensor network can detect danger and direct people along safer paths to exits [9].

These many possibilities have accompanying challenges. Moore's Law means that wireless sensor nodes can — and will — be tiny, a few millimeters on a side. Their energy sources, however, do not have the same governing principles. Chemical energy density limits storage technologies such as batteries or fuel cells, while ambient energy density limits renewable energy sources such as solar panels. A wireless sensor node's energy source determines its form factor: decrease a node's energy requirements, and you can decrease its size.

As deploying sensor networks can be laborious and disruptive, the longer a deployment lasts, the better. This tension – between node form factor and network lifetime – makes energy the defining resource of wireless sensor networks. It influences software, which must incorporate aggressive energy conservation policies. It also influences hardware, which must trade off increased resources against their accompanying energy costs. While improving technology can reduce the energy consumption of computation or storage, networking has fundamental energy limits. Ultimately, wireless networking requires emitting energy into

the environment. There is a tradeoff between the energy cost of transmitting and receiving: a stronger transmit signal can require less processing on the receiver side, while additional signal processing on the receiver side can allow for lower power transmissions.

Long deployment lifetimes place additional requirements on software besides energy conservation. They require that the software running in a network be fault tolerant. The most useful places to sense are often the most difficult to reach. Physically accessing nodes can be disruptive, expensive, or impossible. This means that a node must not only last a long time, it must last a long time unattended. Software cannot assume that a user will reboot the system whenever it acts strangely.

Furthermore, over such long time periods, with such large numbers of nodes, a user can expect even the rarest bugs to manifest. For long term deployments to be possible, nodes must be energy efficient but also recover well from failures. Failures, such as a software crash, can be constrained to a single node. Hardware mechanisms such as watchdog timers or grenade timers can help a node recover from local failures.

However, failures can also be network wide. As nodes execute in response to sensor data and network traffic, ambient network and environmental noise can violate assumptions protocols make or expose unforeseen corner cases, leading to traffic sinkholes [60], infinite routing loops [66], or broadcast storms [79]. Network protocols and application traffic patterns should therefore make as few assumptions as possible on the behavior of the network, and continue to act reasonably even when other nodes act strangely. Simplicity is critical.

Long lifetimes introduce another requirement for sensor network deployments: in-situ retasking. As a network collects data, users gain a better understanding of what data is important and what is not. A network's function is not always static: different growing seasons, for different produce, can require changing the operation of a precision agriculture network. In the case of a static system that does not change after deployment, lengthy testing can ferret out all but the most arcane software faults. However, in most networks, retasking is critical to adapt to evolving requirements.

Take, as an example, environmental observation and forecasting networks. Measuring rare phenomena is a common requirement and purpose of this sensor network application domain. The National Oceanographic

and Atmospheric Administration (NOAA) has wireless sensors deployed on buoys to measure variables such as wave height, water level, wind, and flow rates. Normally, these data is collected at a reasonably low rate, to measure long term trends. However, if a tsunami occurs, NOAA would like to retask the network to collect high rate, high fidelity data. Usually, NOAA receives warning of such an event through a global seismic network tens of minutes before it occurs: the network must be retasked on the fly, as reliably as possible, and quickly.

Users need to be able to retask a sensor network once it has been deployed, but retasking rarely takes the form of general reprogramming. Wireless sensor networks are application specific: they are designed and deployed for a particular application domain. For example, researchers at Vanderbilt University have developed a wireless sensor network that can pinpoint snipers in an urban setting [96]; such a network has very different sensors, hardware resources, and software structure than a network designed for monitoring roving zebra herds [57]. Instead of general reprogramming, retasking a deployed network involves reprogramming within the network's application domain. For habitat monitoring networks, this retasking can involve changing sampling rates, adding sampling predicates, or choosing different sensors to sample. For a fire evacuation network, retasking can involve reconfiguring the network, resetting it after a fire, or running tests and diagnostics to verify that it works safely.

This need for application-level programming has led to researchers exploring domain specific languages. For habitat monitoring, for example, the TinySQL language proposes using declarative queries as a network programming interface [70]. Another example is SORA, where users control application behavior by specifying the rewards a node can receive for its application-level actions [73]. Each of these programming models has a large and vertically integrated supporting software runtime: integrating new functionality beyond a few narrow classes of functionality is very difficult. These thin, vertical slices provide a solution for a small subset of sensor network application domains, but do not address basic systems questions of tradeoffs between their various design decisions.

1.2 Problem Statement

This dissertation addresses the intersection of these three requirements: efficiency, fault tolerance, and retasking. Traditional operating systems are unconcerned with the first requirement, and meet the latter two with a user/kernel boundary. By virtualizing system resources such as the processor, memory, and files, a traditional operating system protects separate programs from one another. A fault in a process can cause that process to fail, but others can continue to run unaffected. The OS protects itself from applications, but is itself assumed to be fault free.

Unfortunately, sensor network constraints preclude this approach. Current nodes do not have hardware memory protection, a prerequisite for the user/kernel boundary. While some future, proposed platforms will have virtual memory support – the iMote2 platform has a full ARM core – the accompanying energy costs lead to a much heavier-weight platform, creating a very different class of device than the small embedded nodes of today. The basic tradeoff between capabilities and their energy cost prevents the traditional systems solution – throwing more resources at the problem – from being suitable.

For example, small amounts of RAM lead sensor network operating systems, such as TinyOS [51] and SOS [46], to rely on an event driven execution model: maintaining thread stacks can be expensive. The MANTIS operating system provides a threaded execution model [1], but the costs of this approach have led to limited use. Like any other operating system, sensor OSes are driven by and interact with hardware through interrupts. Instead of imposing a synchronous execution abstraction purely in software on top of the underlying, event-driven hardware, TinyOS and SOS push hardware's event driven model to higher level abstractions. The absence of a hard protection boundary makes a hard concurrency boundary difficult to impose, as code can disable the hardware events that might lead to pre-emption. In the simplest example, application code can disable interrupts and enter an infinite loop, bringing the system to a halt.

Hardware support or not, users retasking sensor networks need an application level programming environment so users can retask the system without worrying about causing a serious failure. This environment has to operate under the tight resource requirements that limited energy imposes. Prior work has proposed a range of programming models, but the question of how to build their underlying runtimes has gone largely

unanswered. The spectrum of domain specific languages and deployment requirements means that many runtimes will be needed, with varying degrees of customization. Stated concisely:

> *Wireless sensor networks need an architecture to build runtimes for application-level retasking. These runtimes must be usable in long-term deployments: they cannot impose a significant energy burden and must be fault tolerant. They must be simple to build: if using the architecture is as much work as a custom solution, its utility is limited. Finally, the architecture must be flexible enough to support a wide range of programming models.*

The three requirements for a retasking architecture— longevity, simplicity, and flexibility — are not completely independent. They introduce tradeoffs. For example, if increasing architectural simplicity requires additional data storage (e.g., add additional buffering), then it might preclude programming models that have large RAM requirements for their own storage. Additionally, longevity is a metric that cuts across the full range of design options; architectural decisions and energy costs promote some implementation approaches over others. The research challenge is finding the right design point among the range of options, thereby meeting each requirement well without sacrificing the others.

1.3 Thesis

This dissertation argues that virtual machines — software protection — present a compelling and desirable solution to retasking deployed networks. Instead of native code, application level programs are virtual code; instead of the traditional OS abstraction of user land, a sensor network operating system provides ”virtual land.” As sensor networks have a broad range of application domains, a single virtual machine is unattractive: it may be able to do everything, but its generality will prevent it from doing anything well. The virtual machine running on a node needs to be application specific, tailored to the specific requirements of a deployment.

Application specific virtual machines (ASVMs) provide a fault-tolerant retasking environment for application level codes. Users write programs in a language that compiles to the ASVM instruction set, and the mote ASVM executes the instructions. The ASVM defines the encoding that programs take as they disseminate through a sensor network, but allows optimization from above, through ISA customization and

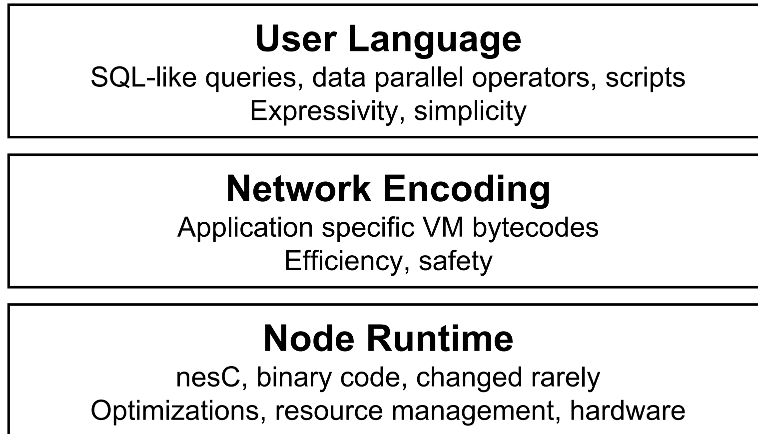


Figure 1.1. The three program layers.

compiler analysis, as well as optimization from below, through run-time transformations, such as just in time compiling. Stated concisely, the thesis of this dissertation is:

> *Application specific virtual machines enable users to efficiently, rapidly, and safely reprogram deployed sensor networks in high level languages while meeting the energy and resource constrains that embedded nodes impose.*

To evaluate this thesis, we designed and implemented Maté, an architecture for building application specific virtual machines for wireless sensor network motes. Maté meets the three requirements of a runtime architecture. It is flexible enough to support a spectrum of programming languages and models, ranging from BASIC to LISP to declarative TinySQL queries. Using Maté, building an ASVM is simple: RegionsVM, a sample ASVM presented in Section 4.6, is a fourteen line text file. Finally, Maté ASVMs are efficient. Compiling to ASVM bytecodes rather than native code reduces the energy cost of installing a program in a network, while customizing the instruction set to an application domain makes their execution energy overhead negligible.

1.4 Maté Overview

Conceptually, the Maté architecture separates retasking into three separate layers: user language, network encoding, and node runtime, as shown in Figure 1.1. The top layer, the user language, is the repre-

sentation in which a user writes a program. This is typically a scripting language, suited to a deployment's application domain. The user language compiles down to the network encoding, which is the form a program takes as it disseminates through the network. The network encoding is the representation that an ASVM can receive over a network interface. The final and bottom layer, the node runtime, is the system that executes the network encoding, using optimizations such as just in time compiling if needed.

The flexibility requirement precludes Maté from defining the user language layer. The architecture takes a position on the node runtime layer — bytecode interpretation — but other implementations could take different approaches while following the same decomposition and using the same architectural concepts. What Maté does specify is the network encoding layer: ASVM bytecodes for codes that follow a certain data and execution model. Defining the middle layer allows flexibility at the user layer while simultaneously enabling the runtime layer to focus on efficiency.

A Maté runtime can be divided into two parts: a common ASVM template and a set of extensions. The template defines the execution and data model, while the extensions determine the ASVM's functionality. A Maté ASVM has an event-driven, threaded architecture. Extensions include the events that trigger the ASVM to execute as well as its instruction set. Instructions can encapsulate application-level operations, such as applying a function over distributed memory, or low level operations, such as arithmetic. Similarly, triggering events can represent application-level phenomena, such as detecting a tracking target, or low level phenomena such as a timer firing. Both are very flexible levels of abstraction.

In addition to meeting the flexibility requirement, this high degree of customizability also enables ASVMs to meet the longevity requirement. Because extensions can be high-level primitives, the in-network representation can be very concise (on the order of a hundred bytes) without sacrificing flexibility. This conciseness reduces both the energy cost of disseminating a program (few packets need to be sent) in addition to the energy cost of executing a program (few instructions need to be interpreted, so time is spent mostly in application logic), while virtualization protects a node from faulty programs.

Finally, while extensions allow users to efficiently retask sensor networks for a wide range of application domains, the template simplifies ASVM composition and implementation. First, as the template structure

allows extensions to be freely combined, the architecture supports there being a library of commonly used extensions such as sensing or communication: a user need only implement the most application-specific ones that are unique to a deployment or set of deployments. Second, composing the extensions and the template into an ASVM is a very simple: the Maté toolchain reads in a simple text file describing the desired extensions and a few other pieces of ASVM metadata. These files can be easily generated by graphical interfaces.

With Maté, a user can easily build a customized runtime that allows him to efficiently, safely, and quickly reprogram a deployed network in terms of high-level application programs.

1.5 Contributions

The contributions of this work are threefold:

- Define the requirements and characteristics of three network retasking layers, identifying the network layer as the critical one in wireless sensor networks;
- Demonstrate that application specific virtual machines are a compelling solution to the issues faced by the network layer, meeting the requirements of flexibility, efficiency, and simplicity; and
- Solve algorithmic and protocol challenges that the in-network layer poses, such as code dissemination and concurrency management.

1.6 Assumptions

The thesis and conclusions of this work make several assumptions about the nature and limitations of wireless sensor networks. We discuss these in further depth in Chapter 2, but they can be briefly summarized:

- **Network activity is the major determinant of a deployment lifetime.**
- **A mote's limiting hardware resource is RAM.**

- **A deployment is inaccessible to physical control or rebooting.**

Chapter 7 discusses the possible effects and research opportunities if technological or research progress were to relax these conditions in the future.

1.7 Summary and Outline

Wireless sensor networks are collections of small, resource constrained devices that must last a long time on limited energy resources. They must also be retaskable, so users can adapt a deployment to changing requirements. This dissertation argues that these requirements call for application specific virtual machines (ASVMs) as a way to safely and efficiently retask deployed networks. ASVMs support the development of domain specific languages, engender in-network optimizations, and greatly decrease the cost of installing new programs. The rest of this dissertation develops and evaluates these arguments more fully.

Chapter 2 describes the current status of wireless sensor networks, including their application domains, hardware, and programming models. These models demonstrate the need for domain-specific programming and establish the functional requirements — the needed degree of flexibility and expressiveness — of the Maté architecture. Hardware platforms define the resource tradeoffs and constraints under which an ASVM must operate. These tradeoffs and constraints influence many aspects of Maté's design.

Chapter 3 decomposes in-situ programming into three distinct layers and defines the requirements of each layer. It compares this decomposition with other common ones, such as shared libraries or the current TinyOS model of whole-application binaries. The three layer decomposition allows ASVMs to codify the middle layer: the in-network representation. Specifying the in-network representation allows optimization from above, as users can program in domain specific languages and customize environments. It also allows optimization from below, as it enables a wide range of ASVM implementations and permits an ASVM to decide how to execute the in-network representation. The need for a customizable in-network representation, combined with the constraints outlined in Chapter 2, establishes the three requirements for an ASVM architecture: flexibility, simplicity, and efficiency.

Chapter 4 presents the idea of application-specific virtual machines as a network layer representation and introduces the Maté architecture. Each subcomponent of the ASVM template and its responsibilities is discussed, as is the structure of ASVM extensions. This chapter evaluates the energy efficiency and resource utilization of two ASVMs, RegionsVM and QueryVM, comparing them with native programs as well as vertically integrated implementations of the same programming models. ASVMs are more efficient than the vertically integrated implementation, show no measurable energy cost over native programs, can represent a wide range of programming models, and are simple to construct.

Chapter 5 presents and evaluates the Trickle algorithm and how the ASVM uses the algorithm in its code propagation protocol. As networking is the principal energy cost to a sensor mote, one of the most important subsystems of an ASVM is its code propagation subsystem. The ASVM template includes a code loading subsystem, which reliably propagates new code throughout a network; if a user installs one copy of a new program, the network will automatically reprogram itself. The scale, distributed nature, and noise in these systems requires that, in order for propagation to be reliable, nodes periodically check whether they have the up-to-date code. The conciseness of ASVM bytecodes – many programs can fit in three or four packets – means that this periodic traffic is a much greater energy cost than the propagation itself. ASVMs achieve rapid propagation while pushing maintenance costs low by using a dynamically adaptive suppression algorithm, called Trickle.

Chapter 6 presents two example ASVMs: SapVM and SmokeVM. SapVM is an ASVM currently being deployed to monitor redwood tree sap flow in Australia. SmokeVM is a prototype ASVM for a fire rescue network. This chapter outlines the application domain of each ASVM and describes the extensions each one includes, demonstrating the flexibility of ASVMs and their feasibility in real-world deployments.

Chapter 7 concludes the dissertation. It discusses the implications of using ASVMs as a programming system and notes several areas of future work in the Maté architecture.

Chapter 2

Background

In this chapter, we present background information on mote networks, including their current application domains, hardware platforms, and software. The constraints and characteristics of mote networks greatly determine the requirements of their programming systems. We overview the basics of the TinyOS operating system and wireless sensor networking, and describe the state of the art in wireless sensor network programming. We touch on sensor network platforms more powerful than motes, to note their differences,

From application domain requirements, we derive three things that a retasking system must provide: a good programming interface, fault tolerance, and energy efficiency. In light of the diversity of sensor network application domains and expected end users, these requirements mean that a retasking architecture must be flexible enough to capture a spectrum of programming interfaces, be simple enough for non-expert users, and produce efficient systems.

2.1 Application Domains

Sensor networks have a wide range of applications and uses. Several concrete examples have emerged in the past few years, either through real deployment or prototype development. In this section, we describe three application domains derived from these examples. The experiences from these examples establish

what form of retasking is needed. This section defines the functional requirements of an in-situ sensor network retasking system.

2.1.1 Periodic Sampling

Periodic sampling is the simplest and most common application domain today. In a periodic sampling application, nodes collect sensor data and possibly deliver that data to a base station using ad-hoc routing. The periodicity can be regular (every five minutes) or irregular (when the user directs it). The data collected can be a single reading from a sensor, readings from a few sensors, or a high frequency time series. Nodes may apply local predicates to determine whether to report data (e.g., "report temperature if it is over 80 degrees"), and the data collected may be at a granularity distinct from a single node (e.g., "report the five highest temperatures in the network").

There are many examples of periodic sampling wireless sensor network applications. Ecology researchers at UCLA are developing a periodic sampling network to measure the effect of rocks on temperature, moisture, and nutrient gradients in arid soils [90]. The Golden Gate Bridge of San Francisco has been instrumented with a periodic sampling network that measures wave forms moving through the structure [62]. On Great Duck Island, a small island off the coast of Maine, wireless sensor networks have measured the microclimate in the nest burrows of the Leach's Storm Petrel in 2002 [103] and 2003 [102].

One recent deployment, a wireless sensor network to measure the microclimate of redwood trees, illustrates the challenges and requirements of this application domain. In the spring of 2004, 80 motes were placed in two 60m tall redwood trees in Sonoma County, California. Every five minutes, each node woke up and sampled its sensors: photosynthetically active radiation, total solar radiation, humidity, barometric pressure, and temperature. The node sent a data packet containing these readings to a collection point (a mote connected to a cellphone) and also logged each sample to non-volatile storage. When not generating or forwarding data, nodes were in a deep sleep state to minimize energy consumption [23].

Logging data turned out to be critical: of the 80 nodes, 65% never successfully delivered packets to the collection point. Additionally, 15% of the nodes stopped generating data after one week. The very low

duty cycle of the network – one packet every 5 minutes – meant the network routing topology adapted and developed slowly. The time it took for the routing topology to develop was long enough that the deployers had to leave before it completed. Additionally, accessing the nodes physically was prohibitively difficult: the deployment was a several hour drive away, and required climbing 60m redwood trees.

In this deployment, the terrible data delivery performance was due to bugs and software faults. 65% of the nodes never delivered data because none of the nodes in one of the trees were ever able to join the routing tree to the collection point. The fixed sampling period prevented the scientists from detecting this was the case before leaving. Being able to inexpensively run tests, execute diagnostics, or fast start the network would have prevented this problem.

The developers hypothesize that an interaction between routing and time synchronization caused the 15% of nodes to die after a week. The interaction prevented those nodes from synchronizing, so they never entered their energy conserving sleep state. The software deployed on the nodes was a large, vertically integrated application. As many aspects of the system were intermingled and dependent, tested subcomponents in isolation was difficult. These dependencies gave the program a very large state space, which testing was unable to completely cover. Unforeseen edge conditions existed in deployment, which happened to emerge and cause 15% of the nodes to die.

2.1.2 Phenomenon Detection

Phenomenon detection is another common application domain. A phenomenon detection network reports high-level application phenomena to a base station or mobile actor within the network. Unlike periodic sampling, nodes collaborate, either locally or globally, to transform these sensor readings into reports about the desired event: there is not necessarily a constant data stream out of the network. Nodes can change their operating model when an event is detected, suddenly moving into a higher energy state where they sample and communicate more frequently. Additionally, while periodic sampling can tolerate latency (the data is processed later, offline), detecting an event is latency sensitive. Finally, while data collection is con-

cerned with reconstructing some signal or environmental field over time and space, phenomenon detection applications seek to detect transient events within those fields [32].

Phenomenon detection applications are more complex and difficult to develop than periodic sampling applications. Compared to the large numbers of periodic sampling deployments, there are fewer phenomenon detection applications and they tend to be prototype demonstrations rather than full deployments. For example, the Pursuer Evader Game (PEG) involved a field of approximately 100 motes that detect an evader vehicle with magnetometers and route its location to a moving pursuer robot [94]. The "Line in the Sand" experimental deployment used passive infrared sensors to have nodes build an automatically configuring tripwire sensor [6]. Vanderbilt University has developed an application that can estimate the range to a sniper by computing the time between a bullet shockwave and muzzle report: a network of sensors can, within a second of a sniper firing a bullet, pinpoint that sniper's location to within a foot [96].

The XSM (eXtreme Scaling Mote) demonstration is a good example of the phenomenon detection application domain. XSM consisted of approximately 900 motes deployed across a square kilometer to detect and distinguish the presence of civilians, soldiers, and vehicles [33]. The spatial scale, combined with the number of nodes, required that deployment be as simple and rapid as possible, little more than pushing a button, dropping the node, and hearing its buzzer go off to indicate it was working properly. As the developers had very little idea of what the network's data would look like once it was deployed, the XSM software included Deluge, a protocol for distributing and installing new OS binaries into the network. Learning from the prior experiences of other sensor network deployments, and considering the dangers of being able to install new binaries, the XSM platform included a hardware protection mechanism called the grenade timer. Unlike a watchdog timer, which can be reset, a grenade timer can only be made to go off sooner. While this forced rebooting can make software much more robust, it does not provide perfect protection. For example, it does not protect the system from errors that happen early in the boot sequence, such as an infinite loop in system initialization.

2.1.3 Fire Rescue

Fire rescue is the third example application domain. Unlike periodic sampling and phenomenon detection, which are broad domains with many examples, fire rescue is a narrow and specialized domain. The World Trade Center disaster of September 11, 2001 motivated investigation and development in this domain. The mechanical engineering department of UC Berkeley consulted with the Berkeley and Chicago fire departments to determine how technology could help fire rescue in future disasters [9]. They concluded that the there are three key ways in which fire rescue could be improved:

- Firefighters should have information on where they are in the building, and on the structural health of their surrounding environment;
- Fire commanders should have information on structural health of the building, and on the location and status of firefighters within; and
- Building inhabitants would benefit from smart evacuation guides, which provide clear indications on the best way to exit the building in the presence of smoke and other environmental noise.

A wireless sensor network can satisfy all three of these requirements: nodes can communicate locally with firefighters, route data to a commander, and actuate exit signs or escape indicators. Based on these observations, researchers at Berkeley developed a prototype board with "traffic light" indicators, with large red, yellow, and green LEDs. The board has a pair of these indicators, to indicate both directions in a hallway, for example. The idea is that people move in the direction of the green lights and away from the red lights. Nodes periodically report their status (e.g., battery voltage). When a node detects a fire, for example through a smoke detector sensor, it broadcasts an alarm in the network, which wakes up, starts reporting data at a faster rate, and illuminates exits.

The operational logic of a fire rescue network is static and unlikely to be changed: it must be tested and certified for safety. However, the network must be administered. It must be configured, so nodes know where exits are and what paths to them exist. Administrators and safety officials need to be able to run diagnostics and test on the network, such as cycling through the indicators to make sure they are functional.

In the event of a real alarm, fire personnel need to be able to remotely reset the network and restore it to a safe state without turning every node on and off. To be useful in a real world scenario, a fire rescue network needs a lightweight and simple programming interface.

2.1.4 Application Analysis

Although each application domain has its own individual requirements and goals, there are several commonalities. A retasking system for a deployment must provide three things:

1. **A retasking system must have a suitable programming interface.** Each sample application domain requires different forms of retasking flexibility. A user of a periodic sampling network needs to be able to adjust sampling rates and what is sampled, while a user of a phenomenon detection network needs to be able to change the phenomenon detection logic, and a fire rescue network administrator needs to be able to reconfigure escape paths and run diagnostics. This range of requirements means that a single programming model would be problematic as a simple interface to all of the possible networks. Instead, each application domain requires its own programming interface, designed for its specific requirements and needed degrees of flexibility.

2. **A retasking system must be fault tolerant.** The ability to dynamically retask a network increases the chances that someone will install a buggy program. A retasking system must protect the node runtime from problematic programs. The most important invariant to maintain is a node's retaskability: as long as a user can always replace the current program, he can recover the system.

3. **Finally, a network with a retasking system must be energy efficient.** A retaskable network must be able to last a long time: the retasking system must not preclude lengthy deployments. Ideally, a network retasked with a dynamic program should have the same lifetime and energy budget as one that is not retaskable. A user should not have to choose between a significantly longer lifetime and the ability to dynamically retask a system.

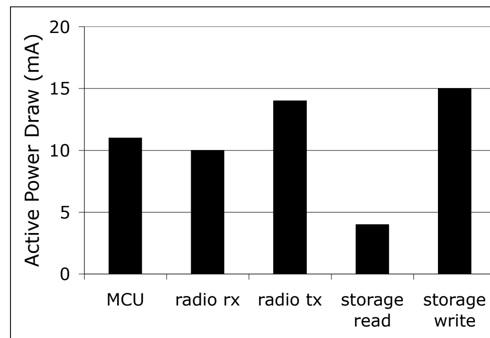
These are the requirements for each retasking system; they say nothing about how that system is built. Building a new system from scratch for each deployment takes a lot of work and testing. Having a general architecture with which to easily build retasking systems that meet these three requirements for a wide range of application domains would enable users to explore design options based on real data, allow fine tuning and adaptation, and transform static networks into dynamic ones that can change their behavior in response to external events. Later in this next chapter, we discuss this issue in greater depth and define the requirements for such an architecture.

2.2 Hardware Platforms

A wide range of mote platforms has emerged over the past five years. The basic tradeoff in platform design is between increased resources and reduced energy consumption. Increasing resources such as memory, CPU, and radio bandwidth can simplify development and enable more complex applications. However, doing so requires either increasing node form factor (larger batteries) or having a shorter lifetime. So far, for mote-class devices, the trend has been to prefer lifetime over resources; by modern standards, motes have tiny resources, on the order of a few kilobytes of RAM and a few megahertz of processor. In this section, we describe the current state of the art in mote platforms. We begin by describing the basic structure of current platforms, using the mica2 as an example. We delve into the two most important subcomponents — the microcontroller and radio — in depth. We distinguish current limitations from fundamental ones, and analyze technological trends by comparing the mica2 with a more recent platform, the Telos revB. This section defines the resource tradeoffs and limitations that retasking systems must operate under.

This section focuses on platforms developed by UC Berkeley in collaboration with the Crossbow corporation. There are many platforms besides these, such as the Eyes platform from TU Berlin and the BTnode family from ETH Zurich [12]. We focus on the Berkeley platforms because they are a long progression of designs that illustrate the general hardware trends.

Microcontroller	ATmega128
RAM (kB)	4
Program memory (kB)	128
Speed (MHz)	8
Active power (mA)	8
Sleep power (μA)	19
Wakeup time (μs)	180
Radio	CC1000
Modulation	FSK
Data rate (kbps)	38.4
Receive power (mA)	10
Transmit power at 0dBm (mA)	14
Non-volatile Storage	AT45DB041B
Size (kB)	512
Page size (B)	264
Page read time (ms)	< 0.25
Read power (mA)	4
Page write time (ms)	14
Write power (mA)	15
Page erase time (ms)	8
Erase power (mA)	15
Page write limit	10,0000
Expansion connection	51-pin
Energy source	2xAA
Capacity (mAh)	1700



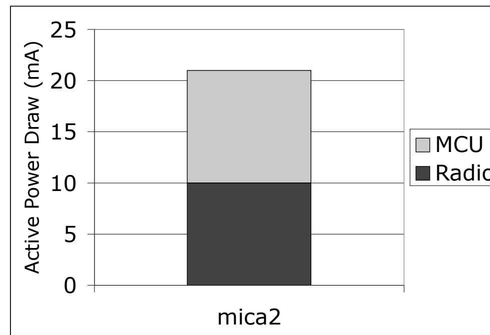


Figure 2.1. Profile of the mica2 platform. The table on the left shows the complete breakdown of the subcomponents, their costs, and performance. The chart at the top right shows the relative power draws of the subcomponents when in common operating states. The chart at the bottom right shows the power draw breakdown of a common operating scenario: being fully awake and receiving messages. These numbers represent the costs for the entire platform; for some subsystems, such as the MCU, this includes additional resources beyond a single chip, such as an external oscillator.

2.2.1 An Example Platform: mica2

The mica2 has five basic subcomponents: a microcontroller, a radio, a power source, a non-volatile storage chip, and an expansion connector. Figure 2.1 shows its components and their characteristics. The mica2 power source is generally two AA batteries, and a node is therefore approximately 3.0 by 6.0 cm in size (the size of 2xAA, side by side). Two AA batteries can provide approximately 1700 mAh of current. When fully active, the mica2 draws 25-30 mA, depending on whether the mote is transmitting or receiving; approximately half of this cost is the microcontroller and half is the radio. At this power draw, a mica2 can last for approximately two and a half days of continuous operation on its batteries. In a deep sleep state, it can theoretically last for over five years. Lasting for a long period of time therefore involves deciding how to spread the two and a half days of possible awake time over the desired lifetime.

Figure 2.1 also summarizes the relative power costs of common system operations, such as CPU activity, radio transmission/reception, and interacting with non-volatile storage. Finally, it shows the basic power draw profile of a common active state, when a mote is awake and receiving packets.

2.2.2 Microcontroller

Most mote platforms use one of two microcontroller families, the ATMega or the TI MSP430. The mica mote family – the mica, mica2, mica2dot, and micaZ – uses the ATMega. The first platform in the family, mica, was produced in late 2001, and its successors have, for the most part, left the design unchanged except for the introduction of new radios. More recently developed platforms – such as Telos and Eyes – have tended towards the MSP430 microcontroller family.

Table 2.1 shows the characteristics of the microcontrollers current platform use [26, 24, 25, 54, 55]. There are three basic trends. The first is an initial increase in program memory, which has since stabilized: the 128KB of the ATMega128 proved to be much more than any application used. The second is a tenfold increase in RAM over the past five years. This growth is due to application pressures: as motes operate at very low duty cycles, greater CPU cycles are not helpful, except to handle time-critical interrupts, which are

<table>
<tr><th></th><th>ATMega 163</th><th>ATMega 103</th><th>ATMega 128</th><th>MSP430 F149</th><th>MSP F1611</th></tr>
<tr><td>Date of Introduction</td><td>2000</td><td>2001</td><td>2002</td><td>2004</td><td>2005</td></tr>
<tr><td>First Platform</td><td>rene2</td><td>mica</td><td>mica2</td><td>Telos revA</td><td>Telos revB</td></tr>
<tr><td>Word Size (bits)</td><td colspan="3">8</td><td colspan="2">16</td></tr>
<tr><td>Address Size (bits)</td><td colspan="3">16</td><td colspan="2">16</td></tr>
<tr><td>Clock Speed (MHz)</td><td colspan="2">4</td><td>8</td><td colspan="2">4.15</td></tr>
<tr><td>RAM (kB)</td><td>1</td><td colspan="2">4</td><td>2</td><td>10</td></tr>
<tr><td>Program Memory (kB)</td><td>16</td><td colspan="2">128</td><td>60</td><td>48</td></tr>
<tr><td>Sleep Current (μA)</td><td>15</td><td>40</td><td>15</td><td>1.9</td><td>2.6</td></tr>
<tr><td>Active Current (mA)</td><td>5</td><td>5.5</td><td>5.5</td><td>0.56</td><td>0.60</td></tr>
</table>

Table 2.1. Microcontrollers used in mote platforms. Left to right, they progress in time from first to most recently used. The power values represent maximum draw and assume a 3V supply voltage and operation at 25° C.

generally not a major issue. Instead, the amount of data and state that an application can maintain between its periods of activity is critical.

The microcontrollers used in mote platforms typically have static RAM (SRAM), whose maintenance requires much less power than dynamic RAM (DRAM). This power reduction requires greater circuit complexity, however (6 transistors instead of 1), and cost. Unlike the radio or the MCU's processing logic, the SRAM draws power when the mote is in its deep sleep state. More SRAM therefore increases the sleep power draw. At very low duty cycles, this can have a significant effect on the lifetime of a mote. For example, the increase in sleep current between the MSP430 F149 and F1611 (1.9 to 2.6 μA) is mostly due to the increase in SRAM from 2 to 10kB. Current low-power SRAM technologies advertise power draws on the order of 10μA/Mbit [72].

This consumption means that it is possible – from a power standpoint – for motes to have more RAM than they do today, but not much more. For example, 64kB would cost on the order of 5μA to maintain in sleep states; maintaining this memory for five years is approximately one quarter of the charge in a AA battery (220 mAh). Correspondingly, maintaining a MB for five years would require 4 AA batteries. Current fabrication and cost issues may preclude larger amount of RAM in the near term. The general trends in microcontrollers is to increase RAM, but power considerations prevent this from continuing to very large amounts: while motes will have more RAM, it will continue to be a valuable resource. In comparison, program memory and compute cycles are reasonably plentiful.

	RFM TR1000	CC1000	CC2420
Date of Introduction	1998	2002	2004
First Platform	WeC	mica2	Telos revA
Sleep Power (μA)	0.7	0.2	20
Receive Power (mA)	3.1	7.4	19.7
Transmit Power at 0 dBm (mA)	12	10.4	17.4
Bandwidth (Kbps)	10/40	38.4	256
Current/bit (nJ)	3600	810	231
Interface	bit	byte	packet
Modulation	OOK/ASK	FSK	O-QPSK
Sleep warmup time (μs)	16	1500	2000

Table 2.2. Radio chips used in mote platforms. Left to right, they progress in time from earlier to more recently used. The power and energy values assume a 3V supply at 25° C.

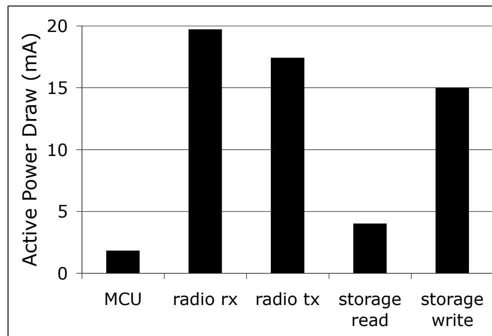
2.2.3 Radio

Table 2.2 shows several radios used in mote platforms. Several early platforms used the RFM TR1000 [89], a very simple radio that provides a bit interface to software, enabling a great deal of research in media access and data link protocols. Over time, however, platforms have tended towards more complex radios that have higher data rates and higher power draws. While power draw has increased, the energy cost to send a bit has decreased. Additionally, the trend towards more complex radio chips has led to an increase in the time it takes to transition from a low power (sleep) to active state. In the case of the CC2420, waking up has the same energy cost as sending 65 bytes of data. Platforms can reduce this time through a variety of means: on the Telos revB, for example, the warm-up time is 500μs [83].

One characteristic not represented on the table is packet loss and range. There have been several experimental studies of the TR1000 and CC2420 at the packet level with specific software stacks [21, 115, 39]. While there are no published studies of the CC2420's behavior, anecdotal evidence suggests that it has a much greater (e.g., 3x) range. This characteristic, combined with its higher data rate, has been the principal motivation towards its adoption.

Unlike microcontrollers, whose energy cost is decreasing, the trend in mote radios is towards more complex chips with higher data rates and correspondingly higher energy consumption. One advantage of this approach is that it simplifies software. The CC2420 provides many features in hardware, such as synchronous acknowledgments and encryption, which were part of the software stacks on top of the simpler TR1000 and CC1000. This energy/resource trend reflects the need for low duty cycle application behavior.

Microcontroller	MSP430 F1611
RAM (kB)	10
Program memory (kB)	48
Speed (MHz)	4.15
Active power (mA)	1.8
Sleep power (μA)	5
Wakeup time (μs)	6
Radio	CC2420
Modulation	O-QPSK
Data rate (kbps)	256
Receive power (mA)	19.7
Transmit power at 0dBm (mA)	17.4
Non-volatile Storage	ST MP25P80
Size (kB)	1024
Page size (B)	65536
Page read time (ms)	< 0.25
Read power (mA)	4
Page write time (ms)	8
Write power (mA)	15
Page erase time (ms)	1000
Erase power (mA)	15
Page write limit	10,0000
Expansion connection	10-pin
Energy source	2xAA
Capacity (mAh)	1700



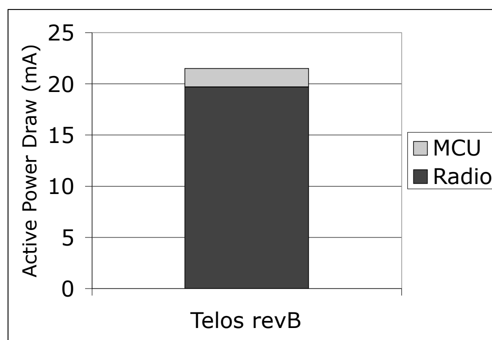


Figure 2.2. Profile of the Telos revB platform. The table on the left shows the complete breakdown of the subcomponents, their costs, and performance. The chart at the top right shows the relative power draws of the subcomponents when in common operating states. The chart at the bottom right shows the power draw breakdown of a common operating scenario, being awake and receiving messages. These numbers represent the costs for the entire platform; for some subsystems, such as the MCU, this includes additional resources beyond a single chip.

Rather than always keep the radio on, a node only turns it on when needed and sends out a flurry of data packets. As long as the amount of data communicated is above a reasonably small value (e.g., 200 bytes), the energy saved by having a lower energy/bit cost makes up for the increased cost of wakeup.

2.2.4 An Example Platform: Telos revB

As one of the most recent mote platforms, the Telos revision B (revB) shows how the trends in technology affect a platform as a whole. Figure 2.2 shows the resources and energy costs of its subcomponents.

Compared to the mica2 (Figure 2.1), the microcontroller consumes a much smaller portion of the energy budget. Compared to communication, processing is inexpensive.

2.2.5 Hardware Trends

The notion of very low duty cycles skews mote resource distributions. Except in rare periods of very high activity, neither CPU nor radio bandwidth are limiting resources for an application, as they spend most of their time inactive. Instead, RAM is usually the resource that applications find most limiting. RAM determines the amount of state that programs can maintain across periods of activity. However, as RAM must be maintained while a node is asleep, increasing RAM increases sleep power draw. This means that RAM will likely remain a limiting resource into the near future.

Microcontroller power draws are decreasing, while radio power draws are increasing. The increase in radio energy cost is due to increased bandwidth and hardware complexity. Communication has always been the principal energy cost in mote platforms, and its dominance is increasing. This pushes application behavior further towards long sleep periods, punctuated by flurries of activity that amortize warm-up and startup costs. Decreasing communication through more complex calculations — using more compute cycles to reduce network activity — is a good general technique to increase network lifetime.

Taken together, these constraints mean that sensor network systems must optimize for two things: reducing RAM footprint and reducing network communication. The former optimization conserves a valuable resource, while the latter increases network lifetime. From the perspective of a retasking system, both optimization pressures share the same solution: minimize code size. Being able to compile dynamic, application-level programs to very dense bytecodes reduces both RAM requirements and the energy cost of propagation.

```
interface BareSendMsg {
  command result_t send(TOS_MsgPtr msg);
  event result_t sendDone(TOS_MsgPtr msg, result_t success);
}
```

Figure 2.3. TinyOS' BareSendMsg interface, which presents the abstraction of sending a fixed sized message buffer (a TOS_MsgPtr). The nesC interface specifies two call directions. A provider of the interface implements the send command and signals sendDone to the caller; a user of the interface can call the send command, and implements the sendDone handler.

2.3 TinyOS

Mote hardware platforms define the resource constraints and tradeoffs that a retasking system must operate under. However, there are further considerations in the system design: the underlying operating system, its structure, and its abstractions. In this section, we outline TinyOS, a popular sensor network OS designed for mote class devices [51], which the Maté architecture is built on top of.

TinyOS provides two basic abstractions: a component-based programming model, and a low-overhead, event-driven concurrency model. We introduce nesC [41], the language used to implement TinyOS and its applications. Mote networks have very different usage models and requirements than more traditional Internet-based systems, and correspondingly different networking abstractions. We describe the general structure of the TinyOS networking stack, as it influences the design of a retasking system's protocols.

TinyOS's event-driven concurrency model does not allow blocking operations. Calls to long-lasting operations, such as sending a packet, are typically split-phase: the call to begin the operation returns immediately, and the called component signals an event to the caller on completion. Unlike most systems that depend on callbacks (e.g., GUI toolkits), in TinyOS/nesC programming these callbacks are bound statically at compile time through nesC interfaces (instead of function pointers passed at run-time), enabling better analysis and checking.

2.3.1 Components and Interfaces

Components are the units of nesC program composition. Every component has two parts, a *signature* and an *implementation*. The signature specifies the interfaces that a component provides and uses. Interfaces are a *bidirectional* relationship: not only can a user invoke code in the provider of an interface, the provider can also invoke code in a user. This is to support the split-phase operations, such as sending a packet. Figure 2.3 shows the `BareSendMsg` interface, which has both a command and an event. A component that uses `BareSendMsg` can call the `send` command, but must implement the `sendDone` event, while a component that provides `BareSendMsg` must implement the `send` command, but can signal the `sendDone` event.

In the example code in Figure 2.4, both the `App` and `Service` components are *modules*. Modules are components with an implementation: nesC code, variables, function definitions, etc. Variables and functions defined within a component can only be named by that component: all inter-component calls are through interfaces.

In addition to modules, nesC has *configurations*, which compose components into higher level abstractions. Like a module, a configuration has a signature: it can provide and use interfaces. However, a configuration's implementation is in terms of other components and their composition. A configuration's interfaces are interfaces of its internal components, exported through wiring.

Consider the examples in Figure 2.5, taken from the Maté architecture. In Maté, every ASVM instruction is implemented as a component which provides the `MateBytecode` interface. The ASVM uses the `MateBytecode` interface to execute ASVM instructions. The module `OPdivM` implements the divide instruction. It depends on being able to access execution state (such as getting its operands). Correspondingly, `OPdivM` uses the interface `MateStacks`.

By itself, `OPdivM` is incomplete: to be a self-contained unit, it needs to be wired to a provider of `MateStacks`. The `OPdiv` configuration does exactly this. `OPdiv` exports `OPdivM`'s `MateBytecode` with the = operator, but wires `OPdivM.MateStacks` to a provider component, `MStacks`, with the `->` operator.. The `OPdiv` configuration represents the completed abstraction that an ASVM will wire to.

```
/* At this App component uses the BareSendMsg interface,
   it can call BareSendMsg.send(), but must handle the sendDone()
   event. */
module App {
  uses interface BareSendMsg;
}
implementation {

  void some_function() {
     call BareSendMsg.send(msgPtr);
  }

  event result_t BareSendMsg.sendDone(TOS_MsgPtr msg, result_t suc-
cess) {
    ...
  }
}

/* As this Service component provides the BareSendMsg interface,
   it implements the call to send() and can signal the completion
   event sendDone().*/
module Service {
  provides interface BareSendMsg;
}
implementation {
  command result_t BareSendMsg.send(TOS_MsgPtr m) {
     ....
  }

  void some_function() {
    signal BareSendMsg.sendDone(m, SUCCESS);
  }
}
```

Figure 2.4. A sample pair of modules, App and Service, which use and provide the BareSendMsg interface of Figure 2.3. As both components are modules, they have implementation code, unlike configurations, which are compositions of components.

```
interface MateBytecode {
  command result_t execute(uint8_t opcode,
                           MateThread* thread);
}
module OPdivM {
  provides interface MateBytecode;
  uses interface MateStacks;
}
configuration OPdiv {
  provides interface MateBytecode;
  ...
  components OPdivM, MStacks;
  MateBytecode = OPdivM;
  OPdivM.MateStacks -> MStacks;
}
module MateEngineM {
  uses interface MateBytecode as Code[uint8_t id];
  ...
  result_t execute(MateThread* thread) {
    ... fetch the next opcode ...
    call Bytecode.execute[op](op, thread);
  }
}
configuration MateTopLevel {
  components MateEngineM as VM, OPhalt;
  components OPdiv, OPhalt, OPled;
  ...
  VM.Code[OP_DIV]      -> OPdiv;
  VM.Code[OP_HALT]     -> OPhalt;
  VM.Code[OP_LED]      -> OPled;
}
```

Figure 2.5. nesC examples.

The distinction between modules and configurations engenders flexible and composable programs. Two components that provide the same interface can be easily interchanged. This ease of composition and separation of implementations is critical to making a retasking system easily customizable.

2.3.2 Parameterized Interfaces

In addition to basic interfaces, nesC has parameterized interfaces. In Figure 2.5, the Maté scheduler, MateEngine, uses a parameterized MateBytecode. Essentially, a component can provide many copies of an interface instead of a single one, and these copies are distinguished by a parameter value, which can be specified in wiring. Parameterized interfaces support runtime dispatch between a set of components. As the parameterization is defined at compile time, the nesC compiler can generate a static lookup table, essentially a large case statement. This makes dispatch efficient and does not requires any additional RAM.

Parameterized interfaces have a wide range of uses. For example, TinyOS active messages dispatch through a parameterized interface, and many system abstractions, such as timers, uses parameterized interfaces to distinguish instances of a service. The case of the Maté architecture, it uses a parameterized interface to execute instructions: each instruction is an instance of the `MateBytecode` interface, and the scheduler dispatches on the opcode value.

2.3.3 Concurrency Model

The nesC concurrency model is based on split-phase operations and the notion of a deferred procedure call (DPC) mechanism called a *task*. Tasks run to completion, cannot preempt one another, take no parameters, and have no return value. Tasks are defined within a component, and obey the same access restrictions as commands and events. Because tasks run to completion, the general approach is to make them short, as a long task can cause the rest of the system to wait a long time and reduce responsiveness.

Although tasks cannot preempt one another, an interrupt handler – or code called from an interrupt handler – can.

At the language level, nesC supports the division of tasks and interrupts by defining two classes of func-

tions: synchronous and asynchronous. Synchronous refers to code only callable from tasks; asynchronous refers to code callable by either an interrupt handler or a task. By default, all functions are the former. A programmer must explicitly define a function as asynchronous with the `async` keyword. An interface specifies which of its commands and events are async:

```
interface ADC {
  async result_t command getData();
  async result_t event dataReady(uint16_t data);
}

interface SendMsg {
  result_t command send(uint16_t addr, uint8_t len, TOS_MsgPtr msg);
  result_t event sendDone(TOS_MsgPtr msg, result_t success);
}
```

The distinction of synchronous and asynchronous code allows nesC to perform compile-time data race detection. The semantics are that all synchronous code runs atomically with respect to other synchronous code. That is, synchronous code does not have to worry about concurrent execution of other synchronous code. This does not mean that synchronous code is never reentrant, due to the bidirectional nature of interfaces. For example, an application handling a reception event from a networking component can, in the receive handler, call a command on the networking component to send a message:

```
event TOS_MsgPtr ReceiveMsg.receive(TOS_MsgPtr msg) {
  ...
  call SendMsg.send(...); // Calls component signaling receive
  ...
}
```

From the perspective of the networking stack component, it is in the middle of a function call (`ReceiveMsg.receive`) when a function is called on it (`SendMsg.send`).

While the TinyOS concurrency model allows fine-grained parallelism with very few resources, it is not necessarily a simple and intuitive programming model to a novice programmer. While a retasking system can take advantage of the concurrency model for its own implementation, it needs to be able to provide other concurrency models to programmers.

2.3.4 Networking

Mote networks have very different usage models and requirements than the Internet. Instead of a single, unifying network (multihop) protocol, they use a wide range of network protocols. End-to-end flows are not the basic networking primitive. Instead, the need for many network protocols pushes the unifying abstraction one layer lower: the data link (single hop) layer.

The basic TinyOS packet abstraction is an Active Message [108]. AM packets are an unreliable data link protocol; the TinyOS networking stack handles media access control and single hop communication between motes. Higher layer protocols (e.g. network or transport) are built on top of the AM interface.

AM packets can be sent to a specific mote (addressed with a 16 bit ID) or to a broadcast address (`0xffff`). TinyOS provides a namespace for up to 256 types of Active Messages, each of which can each be associated with a different software handler. AM types allow multiple network or data protocols to operate concurrently without conflict. The AM layer also provides the abstraction of an 8-bit AM group; this allows logically separate sensor networks to be physically co-present but logically invisible, even if they run the same application and use the same radio frequencies. AM packets have a reasonably small payload. The default is 29 bytes, but programs can increase this if needed. However, as the TinyOS networking stack uses buffer-swap semantics on packet reception, increasing the payload size increases the packet size for every buffer in the system, which can have a large cost.

Unlike Internet systems, which are based on a uniform network protocol, mote networks can have a wide range of network protocols. One example is converge-cast or collection routing, where nodes assemble into one or more ad-hoc trees that stream data up to a collection root [114]. Another is flooding, where the network makes a best-effort attempt to deliver a piece of data to as many nodes in the network as possible. The simplest form of flooding – immediate rebroadcasting – has several well known problems [79, 39]. Correspondingly, more robust flooding protocols introduce suppression timers [36], probabilistic rebroadcasts [14], or other techniques.

A retasking system needs a network protocol to disseminate new programs into a network, and TinyOS defines the abstractions the system has to implement its protocol. Also, applications are built on top of many

different protocols, which the retasking system must therefore be compatible with. For example, periodic sampling commonly uses converge-cast, while fire rescue requires flooding to quickly alert the network when there is a fire.

2.3.5 TinyOS Analysis

The nesC/TinyOS abstractions of components and interfaces are very well suited to designing an extensible architecture. Interfaces precisely define the ways that parts of a system interact. Components are the units in which functionality is implemented, extensibility is added, and users can select between several possible options. The existing TinyOS abstractions, such as timers and active messages, define what an architecture can build on top of, and correspondingly what it must provide.

The TinyOS concurrency model – no blocking operations, tasks, and sync/async – is well suited to low-level systems programming, but not necessarily to application-level programming, which needs higher level abstractions and constructs. For example, while the TinyOS concurrency model is very lightweight, it provides very little protection to the system; a task that has an infinite loop will prevent any other tasks from running and effectively crash a node. Unless a retasking system provides mechanisms for users to exhaustively test code before they install it – a burden which it is, to some degree, trying to alleviate – these problems undercut fault tolerance. Additionally, domain specific languages have a variety of program concurrency models. For example, regions programs are based on synchronous threads while TinySQL queries are abstract declarative statements. A retasking architecture must therefore provide a richer and more robust concurrency model than that of TinyOS.

Programming models may build on top of a wide range of networking protocols. A retasking architecture must be able to express as many of them as possible, so that it does not preclude broad classes of domain specific languages. However, while a retasking architecture should be able to support many network protocols, it should not depend on them. Many protocols are platform or radio dependent; for example, the MintRoute collection protocol is intended for mica2 nodes with a CC1000 radio, while the MultiHopLQI collection protocol only works on platforms that have a CC2420 radio. Remaining network protocol inde-

pendent keeps an architecture platform independent, improving the architecture's portability and making it applicable to a wider range of application domains.

2.4 Architectural Requirements

A sensor network system that allows in-situ retasking has three needed properties. First, it must provide a programming interface to the network that is suitable to the network's application domain. Second, it must be fault tolerant, protecting the network from buggy or ill-conceived programs. Finally, it must be energy efficient enough to not significantly affect a network's expected lifetime.

These are the requirements that every retasking system must meet. An architecture for building these systems has related but distinct requirements, as it must be encapsulate a wide range of them: flexibility, simplicity, and efficiency.

When considering the diversity of sensor network application domains, the first requirement – a suitable programming interface – means that retasking systems for different networks may not share the same language or programming model. A retasking architecture must be *flexible* enough to capture a wide spectrum of code libraries, programming languages, and programming models. System designers need to be able to include functions that represent application-level abstractions. They also need to be able to pick a programming language that is well suited to the application domain and the end users of the network. Finally, the scope of possible languages should not be limited to a particular concurrency model: the underlying system's concurrency model must be general and flexible enough to represent a wide range of language-level ones.

From the perspective of an architecture, the second and third requirements for a retasking system – fault tolerance and energy efficiency – are two manifestations of the same property: a long system lifetime. At a high level, not meeting these requirements has the same result. An system that wastes energy shortens a network's lifetime, causing it to become inoperable earlier than other implementations would. Similarly, a system that can fail irrecoverably causes nodes to stop working earlier than other, fault-tolerant, implementations would.

Finally, building a retasking system with the architecture must be simple. If building something with the architecture is as difficult as building it from scratch, then a user does not gain very much from actual use of the architecture. Separating the common from the domain-specific functionality of retasking systems would allow an architecture provide those commonalities to each system with no effort from the user. For example, every retasking system needs to be able to disseminate new code through a network; providing flexible and efficient implementations of these kinds of services would simplify the designer's job and make the architecture a more effective tool.

In summary, an architecture for building retasking systems must meet three requirements.

1. **Simplicity:** Building a retasking system must be simple. The architecture must be easy to use.
2. **Flexibility:** The architecture must be able to capture a wide range of languages, applications, programming models, and algorithms.
3. **Long lifetime:** The systems the architecture produces must be usable in long-term deployments, being both efficient and fault-tolerant.

2.5 A Flexible Network Encoding

The basic question that arises in such an architecture is defining the boundary between user-level programs — *dynamic code* — and the retasking system — *static code*.

As static code is installed rarely (usually only at deployment time) it can be highly optimized native code. As the mechanism for network retasking, users may install new dynamic code reasonably often. Minimizing dynamic code improves system energy efficiency and reduces system resource utilization: it costs less energy to disseminate and takes up less RAM. To meet the longevity requirement, an architecture must minimize dynamic code size.

However, application domain diversity means that networks, as a whole, require many different levels and kinds of abstractions to be available to dynamic code. For some application domains, such as habitat

monitoring, the desired sensing abstraction is a raw reading; others, such as vehicle tracking, need higher level filters and abstractions such as "vehicle detected." Having a hard boundary between dynamic and static code across all retasking systems either precludes the low level operations many domains require (the boundary is too high) or requires a lot of dynamic code to present the right abstractions for many domain (the boundary is too low). To meet the longevity requirement and the flexibility requirement, the architecture must provide a way to tailor the static/dynamic boundary to each application domain.

The thesis of this dissertation is that using application specific virtual machines (ASVMs) is the best way to meet these requirements. Virtualization provides fault-tolerance, an important component of longevity. A virtual machine separates the dynamic code representation from the underlying system, providing a great deal of freedom for optimization. This freedom allows an architecture to make the virtual machine application-specific by tailoring the boundary between dynamic user code and the static code of the VM itself. Each ASVM has its own tailored boundary; while an individual ASVM is well suited to only a single application domain, they are, as a whole, well suited to a wide range of domains.

In the next chapter, we investigate, at a high level, how a retasking architecture should be structured into three separate layers: user program, network encoding, and runtime. ASVMs fill the role of the middle layer: as they are the dynamic code execution engine: they specify the representation codes take as they propagate through a network. Specifying the network encoding keeps the flexibility and freedom needed in the domain diversity of mote networks: ASVMs can support a wide range of programming languages above, and a wide range of runtime techniques below. In the following chapter, Chapter 4, we present and evaluate the Maté architecture, which follows this structure.

Chapter 3

Three Program Layers

In this chapter, we describe a systems-level decomposition of retasking a sensor network, breaking the problem into three separate parts: the user language, the network encoding, and the node runtime. We begin by analyzing current network programming models, not only for wireless sensor networks but also for Internet systems. Based on characteristics of these prior approaches, we conclude that a three layer decomposition is the best fit for sensor networks.

We define the requirements of each layer and how they interact. We argue that in sensor networks the critical design point of the decomposition is the network encoding. However, unlike generic systems, such as the Java Virtual Machine or Microsoft's Common Language Runtime, in sensor networks the middle layer needs to be customizable in order to be efficiently application-specific. Instead of a common virtual machine as the node runtime, like a JVM or CLR, sensor networks use an application specific virtual machine. Finally, we present an energy performance model for a retasking system.

3.1 The Three Layer Model

The problem of retasking a deployed sensor network can be divided into three parts:

- the user language,

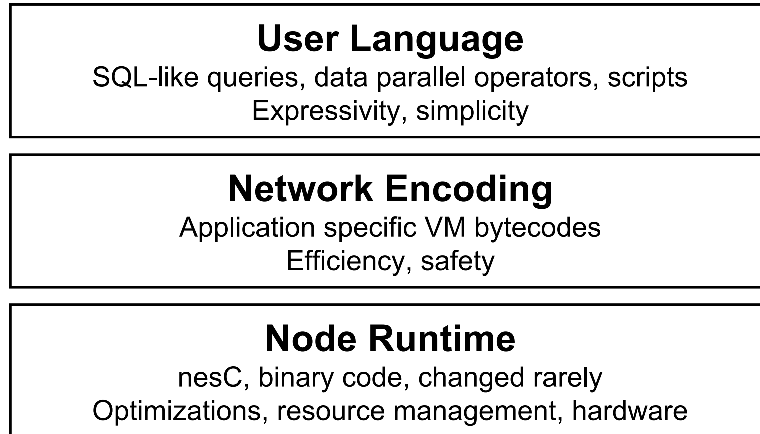


Figure 3.1. The three program layers.

- the network encoding, and
- the node runtime.

Figure 3.1 shows this decomposition graphically. A user writes programs in a user language. The language compiles to the network encoding (dynamic code), which is the representation the network uses to disseminate and forward code. The node runtime (static code) is responsible for taking a network encoding and executing its program.

Loading and executing programs is an old problem – as old as operating systems. In this section, we present several example solutions to this problem, some of which are used in wireless sensor networks, and some of which are used in other system classes. We broadly cluster the examples into three classes: one layer, two layer, and three layer.

The one layer model combines all three parts of network retasking into a single layer. A user writes a program directly against operating system abstractions, typically in a low-level systems language. The program compiles into a native binary image that has complete control of the hardware. Therefore, in the one layer model, a system can run at most one program at a time.

In the two layer model, two of the three parts have been combined. For example, The top two layers might be combined: the network encoding is application source code. The bottom two layers might be

combined: a user writes code in a succinct high-level language, but this compiles to a native binary images that has complete control of the hardware, or a small binary that links against a node runtime.

In the three layer model, the user program compiles to an intermediate representation, such as virtual machine bytecodes, which are the network encoding. Multiple languages might compile to the same network encoding. This representation determines the program to execute, but the node runtime is given freedom on how to execute it. For example, in the case of VM bytecodes, a runtime can translate them into native code using just-in-time compilation techniques.

Based on this decomposition, we examine a spectrum of retasking approaches. We describe the strengths and weaknesses of each approach, and from them distill a decomposition for retasking deployed sensor networks.

3.1.1 One layer: TinyOS

Section 2.3 presents the TinyOS programming model in detail. The TinyOS compilation process combines all code, from application level logic to low level drivers, in an optimized binary image, which is typically 10-40kB in size. One one hand, this allows TinyOS to have very efficient code; on the other, as it enforces a single layer approach, it makes installing new programs an all-or-nothing affair.

The Deluge system, which is becoming a standard part of TinyOS, propagates binary images through a network. Motes have a static set of of program *slots* for binary images. A user can install a binary image into one of the slots, replacing the image that was there before. This involves disseminating tens of kilobytes of data across a multihop network and writing it to non-volatile flash, a significant energy cost. Additionally, as the image is responsible for the entire system, it is easy to introduce low-level bugs that can completely disrupt system behavior.

3.1.2 Two layers: SNACK and Regions

The Sensor Network Application Construction Kit (SNACK) builds on top of the TinyOS component model, providing a higher level component composition language that compiles to a TinyOS program [45].

SNACK code specifies per-node behavior as TinyOS does, but its composition language (the analogy to nesC wiring) has a range of more flexible connection specifications. The SNACK libraries take advantage of this flexibility, providing a data source and sink model. This higher-level language allows users to very succinctly write applications in a limited class of application domains.

Regions is a recent proposal to simplify sensor network programming by providing an abstraction of region-based shared tuple spaces, on which nodes can operate [111]. Users write programs in C-like pseudocode, which compiles to a full TinyOS image. Regions can be based on geographic proximity, network connectivity, or other node properties. In the proposed programming model, TinyOS provides a single synchronous execution context (a "fiber"), which supports blocking operations. While region's constrained memory model precludes memory-based faults, its ties to the TinyOS concurrency model allows a program to render a system unresponsive.

Both Snack and Regions follow a two-layer model, where a higher level language compiles to a full system binary. As with TinyOS, they can use a system such as Deluge to install programs in a network. As Regions depends in part on the TinyOS concurrency model, it can introduce program logic that will cause nodes to fail (e.g., enter infinite loops). In contrast, as it is much more abstract, the SNACK composition language cannot introduce such faults.

3.1.3 Two layers: SOS

SOS is a mote operating system that allows incremental binary updates [46]. It is a two-layer model, as the execution and network representations are the same: users write SOS components in C and compile them into small binaries which link against the SOS kernel. SOS components can also call one another. SOS disseminates components using a variant of the MOAP protocol [99]. Component-level programming reduces the amount of data the network must disseminate to retask. However, the binary code has all of the same safety and robustness issues as TinyOS images, plus more due to dynamic linking making verification harder.

3.1.4 Two layers: Sensorware and TinyDB

Sensorware represents the other side of the two-layer model: users write Tcl scripts, which are sent compressed to nodes, which decompress and interpret them [15]. On one hand, this leads to very small programs: compressed high-level scripts are on the order of 200 bytes long. On the other, it forces a node to have a full Tcl interpreter. Sensorware is intended for nodes with greater resources than motes (such as iPAQs), for which this cost is not significant. On motes, however, it is unfeasible (the interpreter is 179kB of code).

The TinyDB system provides a programming model of declarative, whole network queries in an SQL dialect. The TinyDB system schedules data gathering and communication based on the query. TinySQL supports a family of network aggregates; in addition to basic queries for data from each node, a user can request data from the network as a whole. TinySQL queries are streams of data.

For example, a user can request each node's light value every minute with the query `SELECT light INTERVAL 60s`: this will result in a stream of k tuples per minute (barring packet loss), where k is the number of nodes in the network. In contrast, the query `SELECT avg(light) INTERVAL 60s` will return one tuple per minute, containing the average light value across the entire network. TinyDB uses in-network processing to aggregate data as it is produced, conserving energy.

TinyDB is a two-layer system which merges the top two parts: the network encoding is a binary encoding of a TinySQL query, which nodes interpret. On one hand, this enables high-level declarative queries to have a very concise representation; on the other, it forces nodes to interpret the declarativeness.

3.1.5 Three layers: Java and CLR

Java is a three-layer system: Java is a type-safe object-oriented language that compiles to bytecodes for the Java Virtual Machine (JVM) [44]. The JVM is responsible for executing the bytecodes. While the JVM was initially intended only for the Java language, many languages now have compilers that produce Java bytecodes [106]. Microsoft's CLR is similar to Java, except that its network encoding is intended to be more language neutral [82].

As it is a three-layer system, Java separates the concerns of the three distinct layers. The Java language is intended to be a OO language that is syntactically very similar to C, but with memory safety; it is intended to be easy to learn and be an effective language for general applications. The Java class file format (what the JVM loads) is intended to be reasonably small, by desktop and server standards: they are typically 1-5kB in size. Reducing class file size reduces the time it takes a client to download a Java program. A JVM typically includes a large library of common functionality, so only the application logic is sent over the network. Class files are also verifiable, in that a JVM can determine that a class does not violate Java safety guarantees at load-time.[1] As the bottom layer, the JVM itself is free to use techniques such as just-in-time compiling to improve execution performance.

3.1.6 Analysis

The one-layer model is fundamentally unsuited to application-level retasking. First, it is expensive: small changes require propagating large binaries. Second, it is unsafe: a bug can easily cause the system to fail. Finally, it involves low-level system code, which is at a level of abstraction far lower than is suitable for application users. A biologist who wants to change the rate that sensors are sampled cannot be expected to write and compile low-level TinyOS code.

The two-layer models improve on this somewhat, but still have issues. Systems that conflate the bottom two layers, such as Regions and SNACK, provide a higher-level programming interface, but force the user to install a whole system binary, which is expensive and unsafe. SOS reduces the size of installed updates, but its safety issues remain. Sensorware's and TinyDB's interpretation of source programs allows retasking code to be very concise, but forces a lot of logic into a node, which can consume a great deal of resources and be inefficient.

Existing three-layer models are a step in the right direction, but we claim that they are insufficient. Their limitations lie in enforcing a rigid network encoding. Because Java and the CLR are intended to be general computing platforms, the rigid boundary requires them to have a reasonably low level of abstraction, making

[1] While JVMs can theoretically verify safety properties, in practice JVMs often do so incompletely, leaving type safety holes or security flaws [97].

even simple codes kilobytes in size. While this degree of resource utilization is fine for most systems, it is beyond what sensor motes can provide.

The flexibility enabled by separating user languages from the network encoding provides several benefits. A single language has many possible network encodings, and many languages can share a common encoding. Depending on how a language is used, some encodings may be more efficient than others. Additionally, being able to fine tune or change the network encoding provides flexibility in the underlying runtime. For example, SQL-like declarative queries are a convenient and simple programming interface for basic data collection. One way for a network to support these queries is have declarative queries as the network encoding and push the query planning into each node in the network. Another option is to use an imperative program as the network encoding and have the compiler be responsible for the query plan.

Separating the network encoding from the runtime allows a node to make optimization decisions separate from application-level logic. Depending on its resource tradeoffs, the runtime is free to transform the network encoding into a more efficient representation (e.g., just-in-time compiling if CPU is valuable, or compression if RAM is especially scarce).

This core argument of this dissertation is that sensor networks need a three-layer model in which the middle layer — the network encoding — is customizable, so it can meet application requirements. A customizable network encoding allows a retasking system to take advantage of the common wisdom of programming deployed sensor networks with simple and concise domain-specific languages. Rather than compile these short, high-level programs to a verbose low-level representation, a flexible network encoding allows them to be easily compiled to a customized encoding that better meets the resource constraints sensor networks nodes have. This pushes the application-specific logic into the runtime itself, where it can be efficiently executed as native code.

3.2 Decomposing the Three Layers

Each of the three layers of network retasking has its own requirements and optimization goals. Separating the concerns of each layer allows each one to be designed and optimized independently. Mote platforms

can have different runtime implementations, which take advantage of specific resource tradeoffs or capabilities. A user can pick a programming model that fits the application domain at hand. Finally, the network encoding can be designed to be as concise as possible while maintaining the needed programming flexibility.

3.2.1 User Language

The user language is the retasking interface the network provides, and is hopefully well suited to the network's application. The language needs to have a programming interface that is suitable to the application domain. For example, in a network designed for streaming data collection, such as a habitat monitoring deployment, TinyDB's SQL-like declarative queries might be the user language. In contrast, a network for local event detection and reporting, such as a vehicle tracking deployment, might benefit from an imperative language with library functions for local data aggregation.

Designed for the user, the language has user concerns as its principal requirements. Programs should be simple and succinct: a user should be able to easily express a program for the network. Simple and concise code is shorter and less likely to have bugs. The choice of user language is correspondingly entwined with the application domain of the network.

3.2.2 Network Encoding

The network encoding is the form that a program takes as nodes forward it through a network: it is the data that must be injected into a network for retasking. For example, TinyDB uses binary encodings of SQL queries as its network representation: it encodes SQL keywords and operators as binary values, which TinyDB interprets on a mote. TinyOS programming (or abstract regions, which compiles to a TinyOS program) has a network representation of a binary mote image, which can be installed with the Deluge dissemination system [52].

As nodes transmit programs in the network encoding, its conciseness is very important. The smaller the network representation, the less energy a network must spend to install a new program and the less time it takes for the network to receive it. Unless native code (which can be stored in the mote's program

memory) is used, small network representations also reduce the amount of RAM a mote must use to store a program when sending or receiving it. Network encoding conciseness is therefore an important component of a deployed network's lifetime.

The network representation must not violate the need for fault tolerance. That is, the network encoding cannot allow semantics that can introduce fatal failures. No stream of bits, if presented as a network encoding of a program, should be able to render that node unreprogrammable. Examples of programs that must be prohibited include:

- a program that enters an infinite loop which cannot be interrupted,
- a program that disables interrupts permanently,
- a program that disables networking permanently, and
- a program that corrupts node software state irrecoverably.

Native code is inherently at odds with fault tolerance; in the absence of memory protection, as is the case on mote microcontrollers, a binary program can fail in all four of the ways described above.

3.2.3 Node Runtime

Each node has a runtime, which is responsible for executing the network encoding. It can optimize or translate the network encoding to improve performance as long as it maintains the semantics of the network encoding, protecting the system from aberrant program behavior. The execution representation is how a mote runtime stores a program in order to run it. In some but not all approaches, the execution representation is the same as the network encoding. Examples where this is the case include binary code updates and direct interpreters. Examples where it is not the case include compressed code and systems which support just-in-time compiling.

The node runtime is what actually spends energy to execute and propagate programs. It is therefore responsible for being energy efficient enough to not adversely affect network lifetime. While a concise

network encoding reduces the number of bytes transmitted to propagate programs and the number of bytecodes interpreted, the underlying runtime must provide efficient versions of each of these mechanisms, so the constant factors do not overwhelm the savings from conciseness.

3.3 A Performance Model

Of the three requirements for a retasking system, network lifetime is the most quantifiable. Programming interfaces are subject as much to taste as application domain requirements. The form of fault tolerance that a sensor network retasking system must provide, while possible to evaluate, is mostly an issue of design.

Network lifetime boils down to energy: energy efficiency is a metric that cuts across all of the layers. The user layer must be well suited to the application domain, so programs are succinct; the network encoding must be a concise representation of user programs; finally, the node runtime must propagate and execute user programs efficiently.

In this section, we describe a runtime energy consumption model that enables comparative analysis between different approaches, illustrating the tradeoffs between various implementation decisions. The model divides the energy cost of retasking into two parts: the cost of propagation and the cost of execution. For example, interpreted bytecodes are a more concise network encoding than native code, conserving propagation energy, but their interpretation (or JIT compiling into native code) imposes an execution energy cost over a native binary.

3.3.1 Propagation Energy

The longer a program, the higher the energy cost to propagate it to every node in the network. At a high level, the cost of propagation is linear with respect to the size of the program. In reality, issues such as control packets and loss can complicate this slightly, but these are generally constant factors in terms of the size of the program, or very small considerations. In either case, if the energy cost is greater than $O(n)$, decreasing program size is even more beneficial. In our performance model, we denote the propagation

energy as $P(n) = p \cdot n$, where n is the size of the program in bytes and p is the expected per-node energy cost to propagate a byte of program throughout the network.

Most current propagation services require a low communication rate to maintain that every node is up to date. This overhead is generally independent of the size of a program (it is metadata), and so not a point of comparison between encoding options. More formally, $P(n) = k + (p \cdot n)$, and as k is independent of n it cancels in comparisons.

3.3.2 Execution Energy

Execution energy is the overhead a retasking system imposes on top of actual program execution. A native binary uses all of its energy executing application logic, so in terms of the performance model has no execution energy cost. Using another program encoding introduces an energy overhead, either translating the program to native code (a one-time cost) or interpreting the program (a per-execution cost). The former is simple to analyze. If the cost of translation is greater than the savings from a more concise encoding, then the retasking system is more expensive than disseminating a native binary, although it might provide benefits such as safety. In our performance model, we denote the execution energy as $E(n, t)$ where n is the size of the program and t is time. E depends on how the underlying runtime is implemented.

If the execution overhead is spent in interpretation, then the cost is dependent on not only program length, but also how many times it executes. For the sake of this analysis, we assume that executing a program involves a constant per-byte interpretation cost. This is a simplification, as a program might have loops and program constructs might impose different interpretation overheads. For a runtime that interprets programs, $I(n, x) = i \cdot n \cdot x$, where x is the number of times that the program executes and i is the interpretation's constant factor. If x is based on time (e.g., the program runs periodically), then $I(n, t) = i \cdot n \cdot t$ and $E(n, t) = I(n, t)$. If the number of executions is independent of time (e.g., the program runs once), then x is zero and the overhead can be considered a simple constant.

If the execution overhead is spent in on-node translation/compilation, the complexity of translation determines the execution cost. For example, if translation requires $O(n^2)$ operations, then $C(n) = c \cdot n^2$,

where c is translation's constant factor. The more quickly translation cost grows, the greater benefit to shorter code. This equation assumes that the generated native code is as efficient as if it were hand written and produced by a native compiler, which is unlikely. If not, then there is a per-execution cost that can be modeled as an interpretation cost, so $E(n, t) = C(n) + I(n, t)$. However, the constant factor i of $I(n)$ in this case is probably much lower than pure interpretation.

3.3.3 Energy Analysis

Binary images have no execution overhead (they are the baseline), so their cost is $P(n)$. Other program representations have a cost of $P(n) + E(n, t)$. The cost $E(n, t)$ has time as a parameter for programs that run periodically. Assuming that n is higher for binary code, this means that the energy saved by a more concise program will slowly be consumed by execution. After some period of time, it would have been more energy efficient to send an optimized binary. But both P and E depend on n: decreasing the size of a program reduces both its propagation and execution energy. This is based on the assumption that per byte interpretation energy is reasonably represented as a constant.

Admittedly, similar overhead assumptions have not held in other program encodings. For example, architecture research showed that simple instruction sets (RISC) are more efficient than complex ones (CISC), leading to faster program execution. This is partially due to the complexity of instruction decoding. [13]. However, if the network encoding and underlying runtime can be structured to keep decoding time as close to a constant as possible, then this assumption can hold. Additionally, unlike a hardware platform for general computing, whose complexity must be borne by all applications, in sensor networks additional complexity can be added or precluded on a per-deployment basis.

A given program has two possible energy profiles: the first is a binary, whose cost is $P(n_b)$. The second is an alternative program representation, whose cost is $P(n_a) + E(n_a, t)$, where $n_b \gg n_a$. When $E(n_a, t) > (P(n_b) - P(n_a))$, then the binary program has a lower energy cost. When a user needs to reprogram a network often, the savings of a more concise representation can be dominant. When a user

needs to reprogram a network once and run the program many times, the savings of the efficiency of a binary representation become dominant. This execution efficiency, however, does come at the cost of safety.

3.4 Defining the Network Encoding and Runtime

The user language depends on a variety of factors, including user taste. Even within a particular application domain, different users may prefer different abstractions or execution primitives. There are varying needs across application domains. While SQL is well suited to streaming data collection, expressing local operations is cumbersome and difficult. In contrast, while local broadcasts are a simple way to express local aggregation, they require a lot of code and logic to build a streaming data collection system. The range of applications and corresponding requirements has led to many proposals for sensor network programming models, each suited to a different subset of application domains.

This variety means that no single programming language exists for all wireless sensor networks. To quote Fred Brooks, ”There is no silver bullet.” Instead, there are many, and a reprogramming architecture needs to be able to support as many of them as possible.

3.4.1 The Network Encoding

The network encoding should be distinct from the user program. Programs compile to a general – but customizable – network encoding. This encoding can be optimized separately from a language and underlying runtime. Multiple languages can compile to the same encoding, and there can be multiple runtime implementations for a given encoding. which allows parallel progress and improvement The runtime of a platform can be designed for its resource tradeoffs, while the programming language can be designed for application and user requirements. Specifying the network encoding decouples the layers above from those below, maximizing flexibility.

3.4.2 The Underlying Runtime

Reprogramming a network requires having a runtime for executing and disseminating the network encoding. A runtime is responsible for three things:

- transforming the network encoding into an execution representation,
- executing the execution representation, and
- disseminating the network encoding.

The first two are closely entwined and represent the actual runtime environment. The final responsibility, propagation, deals with how the network installs a program: the runtime is responsible for propagating the network image to all of the nodes that need it. Finally, there is an implicit requirement of flexibility: the runtime must support the required flexibility of the network encoding.

The node runtime and its execution representation depend on two things: hardware capabilities and the complexity of the software that implements it. The hardware capabilities define where a runtime can store the representation. For example, on Harvard architectures a runtime cannot run native instructions out of data memory (RAM). In this case, the runtime must place the native code in instruction memory, which in the AVR series used in the mica family of motes, is non-volatile flash that requires a reboot to access.

Generally, improving the performance of the runtime requires increasing its software complexity. To take an example from Java, the software to directly interpret Java bytecodes is much simpler than the software to JIT bytecodes to native code. Although instruction memory is not a common limitation for applications, complex software systems such as TinyDB have grown large enough for it to be an issue. For example, a common TinyDB build is just over 64KB of program, while the Telos Revision B has 54KB or instruction memory. This and other concerns, such as stability, have led the TinyDB developers to reimplement the system in a simpler fashion.

3.4.3 Runtime Transformation

The node runtime is responsible for taking the network encoding of a program and transforming it into an execution representation. This transformation can be a null operation: in a Java VM that does not use just in time compilation techniques and directly interprets Java bytecodes, the network encoding is the same as the execution representation.

3.4.4 Runtime Execution

The node runtime is responsible for executing the program. It must provide an execution and data model that follows the network encoding's semantics. The runtime is responsible for managing execution parallelism and synchronization. Finally, as the network encoding must be flexible, a node runtime that supports more than one application domain must be similarly flexible. A user must be able to easily introduce new functionality and extend a runtime with the specifics of an application domain.

3.4.5 Runtime Propagation

Finally, the node runtime is responsible for propagating the network encoding within the network. Once deployed, motes often cannot be easily collected; reprogramming cannot require physical contact. Anecdotal evidence from large deployments is that "every touch breaks," that is, every time the nodes must be touched, some percentage (e.g., 5%) fail or break [94]. For a single reprogramming, this may be an acceptable cost. For numerous reprogrammings, it rarely is.

Propagation requires network communication, but networking has a tremendous energy cost. An effective reprogramming protocol must send few packets. While code is propagating, a network can be in a useless state because there are multiple programs running concurrently. Transition time is wasted time, and wasted time is wasted energy. Therefore, an effective reprogramming protocol must also propagate new code quickly. The network runtime must use such a protocol to automatically propagate new code into the network.

3.4.6 Performance Model

In terms of the performance model, the encoding determines n as a function of the desired program. A more concise encoding decreases n. The node runtime defines the functions P and E, depending on the resources of the underlying hardware platform and the algorithms used.

3.5 Maté Bytecodes As a Network Encoding

We claim — and, in Chapter 4, demonstrate — that using ASVM bytecodes as a network encoding provides the programming flexibility sensor networks require without sacrificing performance. With ASVMs, users can retask a network in high-level languages that are well suited to the application domain in question. This high-level code can compile down to tiny bytecode programs, which reduces propagation and execution energy. The principal challenge that an ASVM architecture faces, therefore, is providing mechanisms to define the virtual/native boundary and implement the underlying runtime.

Implementing the underlying runtime — the static code — must meet the architectural requirement of simplicity. Therefore, defining and implementing a native/virtual boundary cannot be difficult. Although ASVMs must be able to support many different user programs and network encodings, they do have common services and abstractions, such as code storage, code propagation, bytecodes, and concurrency management. These commonalities can form a core of functionality that ASVMs share, reducing the work in building an ASVM to just its application-specific parts.

To evaluate the thesis of this dissertation, we have designed and implemented Maté, a general architecture for building and deploying ASVMs. With Maté, a user builds an ASVM by selecting a programming language (in which end-users will program) and a set of domain-specific extensions (e.g., support for planar feature detection). From this specification, the Maté toolchain generates an ASVM (a TinyOS program) as well as a set of supporting tools for compiling programs to the ASVM instruction set and for injecting code into the network.

In the next chapter, we describe and evaluate the Maté architecture. We show that using bytecodes tuned

to the right level of abstraction as a network encoding allows Maté ASVMs to meet the longevity requirement: ASVMs show no measurable energy overhead compared to native programs and are fault-tolerant. We show that being able to tune that level of abstraction allows Maté to meet the flexibility requirement: ASVMs can support very different programming models, such as regions [111] and TinySQL [71]. Finally, building an ASVM is simple: in one case, RegionsVM, the ASVM is a 14-line text file.

3.6 Related Work

Sections 3.1.1 – 3.1.5 gave a brief overview of several retasking decompositions, such as Deluge [52], Sensorware [15], and Java [44]. The diversity of other systems deserves a closer look; simply put, given there are so many existing approaches, why is a new one needed?

For example, while there is little question whether a mote has the resources to run a full JVM or even a reduced functionality KVM [101], recent work at Sun Microsystems has developed a stripped down VM, named Squawk, that can run very small Java programs on a mote [95]. Standard Java classfiles are fairly large, and full bytecode verification can require significant processing. Following the model of Java Card [77], Squawk deals with these problems by introducing an additional compilation stage after the bytecodes are generated by the standard Java toolchain. This bytecode transformation rewrites code sequences to simplify verification and translates large string identifiers into small constants; this affords a 25–70% reduction in code sizes with only minor increases in execution time.

All of these accomplishments aside, whether or not a mote can run a JVM is orthogonal to the greater problem. Java may be a suitable user language for some application domains, but its not for others. Programming in the Java language does not require using a JVM. Finally, while the JVM instruction set may be a good network encoding for some application domains, it is of limited use in others (e.g., declarative SQL queries). The Java architecture constrains a system design unnecessarily; while the rigidity it imposes is useful in the Internet, the requirements of embedded sensor networks require a different solution. Java's limitations stem from the fact that the JVM specifies a hard virtual/native boundary. In theory, an ASVM

architecture should be able to support the JVM bytecode set, although mechanisms such as verification are of limited utility given the single administrative domain model common to mote networks.

Squawk is an example of trying to fit a heavy-weight language into an embedded device. There are also languages designed for embedded devices. For example, FORTH is a stack-based language that requires very few resources [87]. On one hand, FORTH has a lot of good qualities: concurrency, simplicity and composability. On the other, it is a specific low-level programming language, which is inherently unsuited to many forms of application-level programming. While ASVMS can — and do — take a lot of inspiration from FORTH and its implementations, it does not provide the flexibility needed for the diversity of mote application domains.

Introducing lightweight scripting to a network makes it easy to process data at, or very close to, its source. This processing can improve network lifetime by reducing network traffic, and can improve scalability by performing local operations locally. Similar approaches have appeared before in other domains. Active disks proposed pushing computation close to storage as a way to deal with bandwidth limitations [2], active networks argued for introducing in-network processing to the Internet to aid the deployment of new network protocols [105], and active services suggested processing at IP end points [4].

ANTS, PLAN, and Smart Packets are example systems that bring active networking to the Internet. Although all of them made networks dynamically programmable, each system had different goals and research foci. ANTS focuses on deploying protocols in a network, PLANet explores dealing with security issues through language design [50, 49], and Smart Packets proposes active networking as a management tool [93]. ANTS uses Java, while PLANet and Smart Packets use custom languages (PLAN and Sprocket, respectively). Based on an Internet communication and resource model, many of the design decisions these systems made (e.g., using a JVM) are unsurprisingly not well suited to mote networks. One distinguishing characteristic in sensor networks is their lack of strong boundaries between communication, sensing, and computation. Unlike in the Internet, where data generation is mostly the province of end points, in sensor networks every node is both a router and a data source.

The basic issue with these prior approaches is that they establish a hard boundary. For a particular

application domain — such as network protocols — this makes sense: any particular ASVM imposes a hard boundary. However, the issue in sensor networks is that there is a wide spectrum of small application domains, and so an architecture to define those boundaries is needed.

Chapter 4

The Maté Architecture

In this chapter, we describe our application specific virtual machine (ASVM) architecture. We show how using ASVM bytecodes as a network encoding enables application-level programming flexibility without sacrificing energy efficiency. Additionally, by separating an ASVM into two parts — a common, shared template and a set of application-specific extensions — the architecture makes ASVM composition easy: one example, RegionsVM, is generated from a simple fourteen line text file. Finally, the architecture's functional decomposition simplifies extension development and testing, leading to a large extension library.

This chapter describes the ASVM architecture (node runtime) and its network encoding. It evaluates the architecture and encoding with respect to the requirements defined in Chapter 3: energy efficiency, flexibility, and safety. The next chapter goes into one subsystem of the architecture – code propagation – in depth.

Chapter 3 defined the requirements of the network representation of an application program: safety, efficiency, and flexibility. Virtual machines can provide safety. Virtualizing application code separates programs from the underlying hardware and operating system. For example, TinyOS's execution of run to completion tasks and split phase calls forces programmers to use an event driven programming model. A virtual architecture can easily provide alternative execution and data models, such as lightweight threads or declarative queries. Virtualization also has the traditional benefit of platform independent code: different

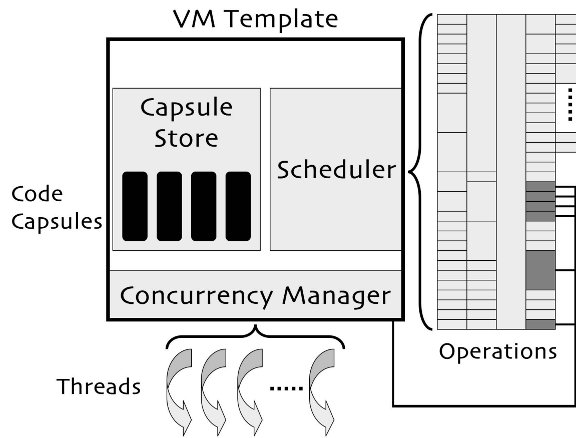


Figure 4.1. The ASVM architecture.

hardware platforms can implement the same virtual machine, allowing a user to reprogram a heterogeneous network uniformly.

As sensor network deployments are application domain specific, a virtual machine can optimize its instruction set with domain specific information. For example, a periodic data collection network needs to be able to sample sensors, set collection rates, save data in non-volatile storage, and route data to a collection point. In contrast, a network that provides smart escape paths in fire rescue needs to configure escape routes, exit points, and monitor fire sensor health. Application primitives that do not change over the life of a deployment can be incorporated into the virtual machine; rather than always including common logic, programs can merely invoke them. This reduces the energy cost of propagation and program memory footprint, satisfying the requirement of efficiency.

Separating application domains into their static and customizable parts establishes a boundary of what parts of an application can be incorporated into the runtime, and what parts need to be dynamically programmed. An ASVM architecture that can efficiently represent a broad range of programming models and application domains satisfies the requirement of flexibility.

4.1 Application Specific Virtual Machine Architecture

Figure 4.1 shows the functional decomposition of an ASVM. The components of an ASVM can be separated into two classes. The first class is the template, a set of components and services that every ASVM includes. The template provides abstractions for execution, concurrency management, code storage and code propagation. The second class is extensions, the set of components which define a particular ASVM instance. An ASVMs extensions include *handlers* and *operations*.

Handlers are code routines that run in response to particular system events, such as receiving a packet, ASVM reboot, or a timer. Operations are the functional primitives of an ASVM. Handlers define when an ASVM executes, and operations define what an ASVM can do when it does execute. Operations specify the ASVM instruction set and built-in function tables. The template and extensions interact through well defined nesC interfaces.

By providing a set of common services and abstractions, the shared template reduces building an ASVM into specifying its extensions. These extensions customize an ASVM to an application domain, providing the needed flexibility for ASVMs to be both efficient and broadly applicable.

4.2 ASVM Template

In this section, we describe the execution and data model of an ASVM, and give a brief overview of its three main subsystems, the scheduler, concurrency manager, and capsule store. Section 4.4 presents the concurrency manager in greater depth, while Chapter 5 covers the capsule store. The section concludes with a description of the template's error handling mechanism.

4.2.1 Execution Model

TinyOS follows a non-blocking programming model to provide high concurrency with limited resources. In this case, the limiting resource is RAM, and the consumer is a C call stack. C call stacks can require a lot of RAM as they must consider the worst case or (in the absence of memory protection or

runtime checks) run over into data memory. Interrupts and a complex call graph lead to large stacks in a conservative, worst case analysis.

In contrast, ASVMs have a threaded execution model that allows blocking operations. Threads are run to completion. As the intention is that programs are very small and succinct, they are unlikely to have long or complex call paths. This means that ASVM threads do not require a great deal of live state. When a thread executes an operation that represents a split-phase call in the underlying TinyOS code (such as sending a packet), the thread blocks. The operation puts the thread on a wait queue, and resumes its execution then the split-phase completion event arrives.

Every handler has an associated thread, which executes the handler's code. At any time, there can only be one invocation of a handler running. If a thread would be triggered again while it is already running, the component that implements it can either store state so it can reinvoke the thread once complete, or ignore the event.

ASVMs are a stack-based architecture: every thread has an operand stack. This operand stack is the only data storage the template provides. Additional storage, such as shared variables, a call stack, or a heap, must be provided through operations, as they are often language-specific.[1]

An ASVM reboots whenever it receives new code. Rebooting involves restoring all running threads to the idle state (halting them), and clearing all script state such as variables. Subsequent events can then trigger handlers to run; the ASVM acts as if it had just booted.

4.2.2 Data Model

The template provides a single operand type: a signed, 16-bit integer. Operations can assume that this type always exists. ASVM extensions can introduce new types, such as a light readings or a linked list; as with storage, the set of types an ASVM supports is often application- and language-specific, so left to a specific ASVM instance.

The architecture separates operand types into two classes: sensor readings and abstract data types

[1] A C memory space is very different than a LISP environment, for example.

```
interface Bytecode {
/* The instr parameter is necessary for primitives
   with embedded operands (the operand is instr
   - opcode). Context is the executing thread. */

  command result_t execute(uint8_t instr,
                           MateContext* context);
  command uint8_t byteLength();
}
```

Figure 4.2. The nesC Bytecode interface, which all operations provide.

(ADTs). This distinction is necessary for storage reasons. Sensor readings are the same size as integers (16 bits) and fit in an operand stack entry. ADTs can of any size, and an operand stack entry for an ADT is a pointer to its value. Because the set of types is language (and possibly ASVM) dependent, but functions are intended to work in any language and ASVM, the architecture requires each added operand type to have a component that provides an interface (`MateType`) for transforming elements of the type between node and network representations. This allows functions to deal with any ADT as a compact region of memory (network representation) for communication or logging, while keeping a node representation for access and modification.

For example, the `bcast` operation locally broadcasts a data packet whose payload is whatever type the program passes. When `bcast` executes, it pops an operand off the stack, converts the operand to its network representation, puts the network representation into a packet, and broadcasts the packet. Receiving ASVMs decode the packet data back into the node representation, so programs can manipulate it. If a program in a language that has a linked list type calls `bcast` with a linked list as a parameter, then the linked list's node-to-network conversion compacts the list into a vector for communication: a receiver unpacks the vector back into a linked list.

4.2.3 Scheduler

The scheduler maintains a ready queue — a linked list of runnable threads – and executes threads in a FIFO round-robin fashion, switching between them at a reasonably fine granularity (a few instructions).

```
interface MateBytecode {
  command result_t execute(uint8_t opcode,
                           MateThread* thread);
}
module OPdivM {
  provides interface MateBytecode;
  uses interface MateStacks;
}
configuration OPdiv {
  provides interface MateBytecode;
  ...
  components OPdivM, MStacks;
  MateBytecode = OPdivM;
  OPdivM.MateStacks -> MStacks;
}
module MateEngineM {
  uses interface MateBytecode as Code[uint8_t id];
  ...
  result_t execute(MateThread* thread) {
    ... fetch the next opcode ...
    call Bytecode.execute[op](op, thread);
  }
}
configuration MateTopLevel {
  components MateEngineM as VM, OPhalt;
  components OPdiv, OPhalt, OPled;
  ...
  VM.Code[OP_DIV]      -> OPdiv;
  VM.Code[OP_HALT]     -> OPhalt;
  VM.Code[OP_LED]      -> OPled;
}
```

Figure 4.3. The OPdivM module implements the divide instruction. The corresponding OPdiv configuration presents a self-contained operation by wiring OPdivM to other components, such as the operand stack ADT (MStacks). The ASVM template component MateEngineM uses a parameterized MateBytecode interface, which represents the ASVM instruction set (the parameter is an opcode value). Wiring OPdiv to MateEngineM includes it in the ASVM instruction set: OP_DIV is the opcode value. MateEngineM runs code by fetching the next opcode from the current handler and dispatches on its MateBytecode interface to execute the corresponding instruction.

The core scheduling loop fetches the running thread's ASVM opcodes from the capsule store and invokes their associated operations through the `Bytecode` interface, shown in Figure 4.2. The interface has two commands: the first, `execute`, executes the operation. The scheduler uses the second, `byteLength`, to know how much to increment the thread's program counter. It increments the program counter before it executes the operation, in case the operation jumps to another address.

Figure 4.3 contains nesC code snippets showing this structure. Resolving bytecodes through nesC wiring allows the scheduler to be independent of the particular instruction set the ASVM implements. Only the scheduler can invoke primitives. Functions, in contrast, can be invoked from outside the scheduler. Functions have a larger scope so ASVMs can support languages with support function pointers, or in which functions are first class values. Every function has a unique identifier, and the scheduler provides access to them through the `Bytecode` interfaces.

The scheduler has two configuration constants. The first is how many operations it issues in an execution of the interpretation task. The second is how many interpretation tasks it issues before switching to the next thread on the ready queue. Increasing the first constant reduces task posting and running overhead, at the cost of forcing other TinyOS components to wait longer. Increasing the second constant reduces queue operation overhead while increasing ASVM thread responsiveness due to the larger execution quantum.

4.2.4 Concurrency Manager

The scheduler maintains the ready queue but the concurrency manager decides which threads can be on the queue. The ASVM template provides mechanisms for implicit thread synchronization that ensure race-free and deadlock-free execution. When the template installs new handlers, the concurrency manager analyzes them for what shared resources they use. It uses this information to enforce implicit two-phase locking. Before a handler can start executing, it must acquire locks on all of the shared resources it may require. As a handler executes, it can release locks, and must release all locks when it completes. Section 4.4 describes the concurrency manager's mechanisms and algorithms in greater depth.

4.2.5 Capsule Store

The capsule store manages code storage and loading, propagating code capsules and notifying the ASVM when new code arrives. Every code capsule has a version number associated with it, and the capsule store always tries to install the newest version it can hear. The capsule store periodically advertises what versions it currently has, so the network can detect and resolve inconsistencies to bring the system up to date. The operating model is that once a single copy of new code appears in the network, the capsule stores will efficiently, quickly, and reliably deliver it to every node. Chapter 5 describes the capsule store, its algorithms, and its protocols in greater depth.

4.2.6 Error Handling

When an ASVM encounters a run-time error, it triggers an error condition using the `MateError` interface, indicating the offending thread, capsule, and instruction. If the error is not associated with an instruction (e.g., a queuing error on system reboot), these fields are empty. When the VM engine receives an error trigger, it halts execution of all threads and periodically advertises the cause of the problem. It stays in this state until new code (which hopefully fixes the problem) arrives. The idea is that errors should be caught early and made explicit, rather than finding data is invalid after a month of deployment.

4.3 ASVM Extensions

Building an ASVM involves connecting handlers and operations to the template. Each handler is for a specific system event, such as receiving a packet. The set of handlers an ASVM includes determine what events it responds to. When a handler's event occurs, the handler implementation triggers a thread to run its code. Generally, there is a one-to-one mapping between handlers and threads, but the architecture does not require this to be the case. The concurrency manager's conservative resource analysis generally precludes several threads from concurrently running the same handler. The one exception is when the handler does not access any shared resources.

The set of operations an ASVM supports defines its instruction set. Instructions that encapsulate split-phase TinyOS abstractions provide a blocking interface, suspending the executing thread until the split-phase call completes. Operations are defined by the Bytecode nesC interface, shown in Figure 4.3, which has two commands: `execute` and `byteLength`. The former is how a thread issues instructions, while the latter is so the scheduler can correctly control the program counter. Currently, ASVMs support three languages, TinyScript, motlle, and TinySQL, which we present in Sections 4.6.1 and 4.7.3.

4.3.1 Operations

There are two kinds of operations: primitives, which are language specific, and functions, which are language independent. The distinction between primitives and functions is an important part of providing flexibility: an ASVM supports a particular language by including the primitives it compiles to. A user tailors an ASVM to a particular application domain by including appropriate functions and handlers. For functions to work in any ASVM, and correspondingly any language, ASVMs need a minimal common data model. Additionally, some functions (e.g., communication) should be able to support language specific data types without knowing what they are. These issues are discussed in Section 4.2.2. In contrast, primitives can assume the presence of data types and can have embedded operands. For example, a conditional jump is a primitive, while sending a packet is a function.

Every operation corresponds to one or more ASVM bytecodes. An operation can correspond to more than one bytecode if it has an embedded operand. For example, the `pushc6` primitive of the TinyScript language pushes its 6-bit embedded value onto the operand stack, and therefore corresponds to 64 (2^6) different bytecodes. The `mba3` primitive of the motlle language is a branch instruction with three bits of embedded operand, which denote the encoding of the address. Operations follow a common naming convention so that the Maté toolchain can build ASVMs and assemble ASVM programs into bytecodes.

4.3.2 Handlers

Every ASVM has a set of handlers, which are the code routines the ASVM runs in response to predefined system events. Operations define what an ASVM can do when it executes, while handlers define *when* it executes. When a handler triggers, it submits a thread to run to the concurrency manager. Handlers are nesC components.

Example handlers include reboot, which runs when an ASVM reboots, receive, which runs when an ASVM receives a packet sent by the `bcast` function, and timer0, which runs in response to a periodic timer started with the `settimer0` function. The Maté framework currently maintains a one-to-one mapping between threads and handlers, to maximize parallelism, but this is not inherent to the execution model.

Some handlers have functions associated with them. Including the handler in an ASVM also includes its functions. For example, handlers representing timers include functions for configuring the firing rate. Similarly, a handler that runs when an ASVM receives a type of packet (e.g., the Broadcast handler) has an associated function for sending those packets (the bcast() function).

4.4 Concurrency Management

This section describes how the ASVM template supports concurrency safety. Sensor network users, more so than other computing domains, needs to be able to reprogram their systems. However, ASVMs provide a challenging execution model: race conditions are common in event-driven thread concurrency. To simplify programming, the ASVM template supports race-free and deadlock-free execution through implicit synchronization. A user can write handlers that share data and be sure that they will run safely.

4.4.1 Implicit Synchronization

Traditionally, the default behavior for concurrent programming environments (e.g., threads, device drivers) is to allow race conditions, but provide synchronization primitives for users to protect shared vari-

ables. This places the onus on the programmer to protect shared resources. In return, the skilled systems programmer can fine tune the use of the primitives for maximum CPU performance.

As event driven execution is prone to race conditions, and sensor network end users are rarely expert systems programmers, Maté takes the opposite approach. When an ASVM installs a new capsule, it runs a conservative program analysis to determine the set of shared resources, such as variables, used by the handlers contained in the capsule. Using this analysis, the Maté concurrency manager ensures race free and deadlock free parallelism. Although a compiler can annotate a given handler with its required resources to simplify a mote's analysis, the exact set of capsules a mote has can dynamically change due to epidemic code dissemination. An ASVM must take care of inter-capsule analysis.

4.4.2 Synchronization Model

The ASVM concurrency model is event-based: a program is specified as a set of handlers for events. The basic unit of execution is an ASVM thread: when an event occurs a thread executes the corresponding handler code. The concurrency manager uses static analysis techniques to infer the resources a thread needs It uses this information to schedule invocations so as to avoid both data-races and deadlocks. Threads can be suspended waiting for some operation to complete (e.g., a message send, or a file read), thus the ASVM model is more elaborate (and easier to use) than pure non-blocking TinyOS. However, the ASVM concurrency manager makes two important assumptions:

- Resumption of a suspended invocation does not depend on actions in other invocations.
- Invocations still follow a run-to-completion model, they are not used to provide "background-task"-like functionality. Invocations, for example, that have infinite loops, can prevent other handlers from running indefinitely.

As a result the programmer is relieved from having to worry about locking resources, proper locking order, etc. And the runtime system has more information on which to base scheduling decisions.

4.4.3 Synchronization Algorithm

The model for resource acquisition and release is as follows: before an invocation can start execution, it must acquire all resources it will need during its lifetime. At each subsequent scheduling point, the invocation can release a set R of resources before suspending execution, and acquire a set A of resources before resuming. To prevent deadlock, we require $A \subseteq R$ (we prove below that this condition is sufficient for building deadlock-free schedulers). Finally, when an invocation exits it must release all held resources. Note that we do not guarantee any atomicity between two invocations of the same handler.

To preserve safety, the static analysis of a handler's resource usage and the runtime system must guarantee that an invocation holds all resources at the time(s) at which they are accessed and that the intersection of the resources held by any two invocations is empty.

As an example, consider a simple ad-hoc routing system on a TinyOS sensor mote. When the TinyOS networking layer receives a packet, it signals an event to an application stating a packet has arrived. The application updates routing tables and enqueues a task to forward the packet. The forwarding task fills in a packet header then issues a command to send the packet. When the packet is sent, the networking layer signals the application that the send has completed. The series of operations performed in response to receiving a packet form an invocation. Three sequences execute: handling the packet reception event, the forwarding task, and handling the packet send done event. Examples of resources are the packet buffer, the routing tables, or the network interface.

Figure 4.4 contains a very simple abstract program which handles a request from a client. It first checks the request for validity – if valid, it uses a resource and sends a response (a scheduled operation). If the request is not valid, it closes the connection to the request (not a scheduled operation). On the left is the program written in our invocation model; on the right is the traditional event-based version. To better show the correspondence between the two versions, we have appended numbers (1) through (3) to distinguish the three `use` statements.

```
handle request
  if valid(request) then
    send response
    use(1) resource X
  else
    use(2) resource X
    close
  end
  use(3) resource Y
end
```

(a) Invocation Model

```
handle request
  if valid(request) then
    non-blocking-send response
  else
    use(2) resource X
    close
    use(3) resource Y
  end
end

handle send done
  use(1) resource X
  use(3) resource Y
end
```

(b) Traditional event-based model

Figure 4.4. Sample Program 1

4.4.4 Deadlock-free

The inequality $A(v_i) \subseteq R(v_i)$ holds at at all scheduling points except start. This inequality makes the set of resources a context holds while active monotonically decreasing over its lifetime. However, some resources are temporarily relinquished during scheduled operations. We show that this condition is sufficient for building a deadlock- and data-race- free scheduler. This deadlock-free scheduler is very simple: every time a context releases some resources, the scheduler examines all waiting contexts. As soon as it finds a waiting context C whose acquire set resources are not in any other contexts's held sets, it has that context acquire the resources and resume. As there is a single scheduler and only the scheduler allocates resources, this algorithm obviously prevents data-races (assuming a correct sequence graph). It also prevents deadlock as this proof shows.

Proof by contradiction: assume deadlock exists. Represent the invocations as a traditional lock dependency graph. Each vertex represents a context. For all vertices C_a in the suspended or waiting state, we record the time t_a at which C_a was suspended (for newly created invocations which start in the waiting state, t_a is the invocation creation time). There is a directed edge from invocation C_a to C_b if C_a is waiting on a lock that C_b holds.

Given our scheduler and invocation model, these edges represent two situations. First, C_a may be a newly created invocation waiting at its start vertex for resources that C_b holds. Secondly, C_b may be holding

```
interface BytecodeLock {
  command int16_t lockNum(uint8_t instr, uint8_t hand-
lerId, uint8_t pc);
}
```

Figure 4.5. The BytecodeLock interface.

onto resources that C_a released after suspending at time t_a but wants to reacquire before continuing. In this second case, because resources can only be acquired atomically at the end of a scheduling point, it follows that C_b resumed execution after C_a was suspended. Therefore, if C_b subsequently suspends at time t_b, we can conclude that $t_b > t_a$. Another property of edges is that I_b cannot be at its entry point: invocations acquire resource sets atomically, so an invocation which is waiting at its start point does not hold any resources.

For deadlock, there must be a cycle $C_1, \ldots, C_n$ in the graph. Every context in the cycle must be in the waiting state. A context C_j from the cycle cannot be at its start vertex as it has an incoming edge, therefore the invocation C_k after C_j in the cycle ($k = 1$ if $j = n$, $k = n + 1$ otherwise) must have $t_k > t_j$. By induction, it follows that $t_1 > t_1$, a contradiction.

4.4.5 Implementation

As languages usually have shared resources, which may have specialized access semantics, the analysis an ASVM performs is language-specific. The most conservative form of analysis assumes all handlers share resources. The motlle language takes this approach, as its Scheme-like semantics would require a full pointer analysis on the mote, which is infeasible. This forces motlle threads to run serially. In contrast, TinyScript considers each shared variable as its own shared resource. As all reads and writes to variables are statically named (there are no pointers), the concurrency manager can easily and inexpensively determine the variables that a handler accesses.

An operation can declare that it accesses a shared resource with *BytecodeLock* interface, shown in Figure 4.5. The interface has a single command, `lockNum`. The concurrency manager uses a parameterized

```
typedef uint16_t CapsuleOption;
typedef uint16_t CapsuleLength;

typedef struct {
  CapsuleOption options;
  CapsuleLength dataSize;
  int8_t data[ASVM_CAPSULE_SIZE];
} ASVMCapsule;
```

Figure 4.6. The ASVM capsule format.

`MateBytecodeLock` interface, where the parameter is an operation's opcode value. The call to `lockNum` returns which shared resource lock, if any, the operation requires.

When the capsule store receives a new handler, it signals this fact to the concurrency manager. The concurrency manager scans through the handler code, storing a bitmask of the shared resource it requires. The concurrency manager stores this bitmask, which is the handler's start acquire set. As a handler executes, it can mark resources for release using the `punlock` operation, or for yielding with the `unlock` operation (the former adds the resource to the release set, the latter to both the acquire and release sets). When a thread executes an operation that constitutes a scheduling point – usually a function that encapsulates a split-phase operation – that operation yields the thread, releasing appropriate resources.

4.5 Program Encoding

A user reprograms an ASVM in terms of code *capsules*. Figure 4.6 shows the ASVM capsule format. A capsule contains one or more handlers. The mapping between capsules and handlers is language specific. For example, in TinyScript, there is a one-to-one mapping (every handler is in its own capsule), while in motlle there is a many-to-one mapping (one capsule contains all of the handlers). A handler is an array of ASVM bytecodes.

4.5.1 Bytecodes

Every Maté instruction has a corresponding nesC component. For example, the `OPhalt` component presents the `halt` instruction, which halts the execution of a thread and releases all of its resources. An operation component is almost always a nesC configuration that wires the operation's implementation to its needed subsystems. `OPhalt`, for example, wires `OPhaltM` to the Maté scheduler and concurrency manager.

As the Maté assembler has to be able to generate binary code for arbitrary instructions, opcodes have a naming convention to specify their width and whether they have embedded operands. All Maté operations have the following naming format:

$$< width >< name >< operand >$$

$Name$: is a string. $Width$ and $operand$ are optional numbers. Width specifies how many bytes wide the instruction is. If no width is specified, the operation defaults to one byte wide. Operand specifies how many bits of embedded operand there are. If no operand is specified, the operation defaults to no embedded operand. Note that, after considering width and operand, the operation must have an opcode in its first byte (Maté can handle at most 256 distinct operations). That is, $1 \leq ((8 * width) - operand) \geq 8$.

These are a few example operations:

Component	Width	Name	Embedded	Description
OPrand	1	rand	0	Random number function
OPpushc6	1	pushc	6	Push a constant onto the stack
OP2jumps10	2	jumps	10	Jump to a 10-bit address

Functions are always one byte wide and never have embedded operands. In the above example, neither `pushc` nor `jumps` could be available as functions; they are primitives. In contrast, `rand` is a function which can be used by any ASVM.

Every bytecode maps to an operation component, which provides its functionality through the MateBytecode nesC interface, shown in Figure 4.3. The set of operations wired to the scheduler defines the ASVM instruction set. Some operations share implementations. For example, the TinyScript language has

```
module OPgetsetvar4M {
  provides interface MateBytecode as Get;
  provides interface MateBytecode as Set;
  ...
}
implementation {
  MateVariable variables[16];
  ...
}

configuration OPgetvar4 {
  provides interface MateBytecode;
}
implementation {
  components OPgetsetvar4M;
  MateBytecode = OPgetsetvar4M.Get;
}

configuration OPsetvar4 {
  provides interface MateBytecode;
}
implementation {
  components OPgetsetvar4M;
  MateBytecode = OPgetsetvar4M.Set;
}
```

Figure 4.7. Loading and Storing Shared Variables: Two Maté Primitives Sharing a Single Implementation

sixteen shared variables, which are accessed with the `getvar4` and `setvar4` primitives. The module allocates the variable state; following the TinyOS/nesC state encapsulation model, this requires both primitives to be implemented by the same component. Each primitive has its own configuration, which wires to the shared module that provides two `MateBytecode` interfaces. Figure 4.7 shows the nesC code for this decomposition.

4.5.2 Compilation and Assembly

Each ASVM language has its own compiler, which produces assembly. The ASVM toolchain provides an assembler for converting assembly instructions into ASVM bytecodes.

When the ASVM toolchain builds an ASVM, it generates the mapping from instructions to opcode values. This mapping is defined in a top-level nesC header file, as a C enumeration (`enum`). The enumeration values are in terms of the operation names: for example, the opcode of the `or` instruction is `OP_OR`. The ASVM toolchain uses `ncg` (the nesC constant generator) to generate a Java class that has these constants as fields. The ASVM assembler is a Java class that takes a stream of assembly instructions and transforms them into the proper binary. As the instruction names state how wide each instruction is, and how wide its embedded operand is (if any), the assembler can transform any ASVM-friendly instruction stream into its corresponding binary representation.

4.6 Building an ASVM

To build an ASVM and its corresponding scripting environment, a user writes a short description file. Figure 4.8 shows the 14 line description file for RegionsVM, an ASVM that supports programming with abstract regions [111]. Section 3 presents this ASVM in greater depth. In RegionsVM, the user has a single handler, which runs when the ASVM reboots; this follows the regions programming model of a single execution thread. A user programs RegionsVM with the TinyScript language, and the ASVM includes several functions that provide sensing, processing, and regions-based communication.

```
<VM NAME="KNearRegions" DIR="apps/RegionsVM">

<LANGUAGE NAME="tinyscript">
<LOAD FILE="../sensorboards/micasb.vmsf">

<FUNCTION NAME="send">
<FUNCTION NAME="cast">
<FUNCTION NAME="id">
<FUNCTION NAME="sleep">
<FUNCTION NAME="KNearCreate">
<FUNCTION NAME="KNearGetVar">
<FUNCTION NAME="KNearPutVar">
<FUNCTION NAME="KNearReduceAdd">
<FUNCTION NAME="KNearReduceMaxID">
<FUNCTION NAME="locx">
<FUNCTION NAME="locy">

<HANDLER NAME="Reboot">
```

Figure 4.8. Minimal description file for the RegionsVM. Figure 4.14 contains scripts for this ASVM.

ASVM description files are composed of XML-like *elements*, such as LANGUAGE, VM, or FUNCTION. An element can have one or more *tags*, which are attribute-value pairs. For example, the FUNCTION elements in the file in Figure 4.8 each have a single tag, the NAME tag.

From the file in Figure 4.8, the Maté toolchain generates four files: a nesC configuration for the TinyOS application, a C header file containing all of the important constants, a text file containing ASVM metadata and documentation, and a Makefile for building the TinyOS application and needed Java classes. There is also a GUI tool for generating VM description files.

The C header file includes bytecode values of VM instructions, code capsule identifiers, configuration constants, such as capsule size and code propagation timers, data types, and error condition codes. The toolchain generates the instruction set by determining the number of bits of embedded operand each operation has and assigning it the according number of bytecode values. The header file also contains the function identifiers which can be used for indirect invocation from outside the scheduler.

4.6.1 Languages

RegionsVM, shown in Figure 4.8, uses the TinyScript language, which is a BASIC dialect. ASVMs currently support three languages, TinyScript, motlle and TinySQL queries. We discuss TinySQL in Section 4.7.3, when presenting QueryVM, an ASVM for in-network query processing.

TinyScript is a bare-bones language that provides minimalist data abstractions and control structures. It is a BASIC-like imperative language with dynamic typing and a simple data buffer type. TinyScript does not have dynamic allocation, simplifying concurrency resource analysis. The resources accessed by a handler are the union of all resources accessed by its operations. TinyScript has a one to one mapping between handlers and capsules. Figure 4.13 contains sample TinyScript code and the corresponding assembly it compiles to. Appendix 9.B contains the full TinyScript grammar and its primitives.

Motlle (MOTe Language for Little Extensions) is a dynamically-typed, Scheme-inspired language with a C-like syntax. Figure 4.9 shows an example of heavily commented mottle code. The main practical difference with TinyScript is a much richer data model: motlle supports vectors, lists, strings and first-class functions. This allows significantly more complicated algorithms to be expressed within the ASVM, but the price is that accurate data analysis is no longer feasible on a mote. To preserve safety, motlle serializes thread execution by reporting to the concurrency manager that all handlers access the same shared resource. Motlle code is transmitted in a single capsule which contains all handlers; it does not support incremental changes to running programs.

A user specifies the language an ASVM supports through the LANGUAGE element. This element causes the ASVM toolchain to load a language description file (`.ldf`) that contains a list of all of the primitives the language compiles to. Figure 4.10 shows the TinyScript description file. Primitives that represent shared resources – which the concurrency manager controls access to – have the `locks` tag.

4.6.2 Libraries

ASVM description files can import libraries of functions. For example, the RegionsVM file uses the mica sensor board library with the LOAD element. The LOAD element reads in the specified file as if it

```
settimer0(500);       // Epoch is 50s
mhop_set_update(100); // Update tree every 100s

// Define Timer0 handler
any timer0_handler() { // 'any' is the result type
  // 'mhop_send' sends a message up the tree
  // 'encode' encodes a message
  // 'next_epoch' advances to the next epoch
  //     (snooped value may override this)
  send(encode(vector(next_epoch(), id(), parent(),
                                   temp())));
}

// Intercept and Snoop run when a node forwards
// or overhears a message.
// Intercept can modify the message (aggregation).
// Fast-forward epoch if we're behind
any snoop_handler() heard(snoop_msg());
any intercept_handler() heard(intercept_msg());
any heard(msg) {
  // decode the first 2 bytes of msg into an integer.
  vector v = decode(msg, vector(2));

  // 'snoop_epoch' advances epoch if needed
  snoop_epoch(v[0]);
}
```

Figure 4.9. A simple data collection query in motlle: return node id, parent in routing tree and temperature every 50s.

```
<LANGUAGE name="TinyScript" desc="A simple, BASIC-like language.">
<PRIMITIVE opcode="halt">
<PRIMITIVE opcode="bcopy">
<PRIMITIVE opcode="add">
<PRIMITIVE opcode="sub">
<PRIMITIVE opcode="land">
<PRIMITIVE opcode="lor">
<PRIMITIVE opcode="or">
<PRIMITIVE opcode="and">
<PRIMITIVE opcode="not">
<PRIMITIVE opcode="lnot">
<PRIMITIVE opcode="div">
<PRIMITIVE opcode="btail">
<PRIMITIVE opcode="eqv">
<PRIMITIVE opcode="exp">
<PRIMITIVE opcode="imp">
<PRIMITIVE opcode="lxor">
<PRIMITIVE opcode="2pushc10">
<PRIMITIVE opcode="3pushc16">
<PRIMITIVE opcode="2jumps10">
<PRIMITIVE opcode="getlocal3">
<PRIMITIVE opcode="setlocal3">
<PRIMITIVE opcode="unlock">
<PRIMITIVE opcode="punlock">
<PRIMITIVE opcode="bpush3" locks=true>
<PRIMITIVE opcode="getvar4" locks=true>
<PRIMITIVE opcode="setvar4" locks=true>
<PRIMITIVE opcode="pushc6">
<PRIMITIVE opcode="mod">
<PRIMITIVE opcode="mul">
<PRIMITIVE opcode="bread">
<PRIMITIVE opcode="bwrite">
<PRIMITIVE opcode="pop">
<PRIMITIVE opcode="eq">
<PRIMITIVE opcode="gte">
<PRIMITIVE opcode="gt">
<PRIMITIVE opcode="lt">
<PRIMITIVE opcode="lte">
<PRIMITIVE opcode="neq">
<PRIMITIVE opcode="copy">
<PRIMITIVE opcode="inv">
```

Figure 4.10. The TinyScript description file.

```
<SEARCH PATH="../../../sensorboards/micasb">
<SEARCH PATH="../sensorboards/sensors">
<FUNCTION NAME="soff">
<FUNCTION NAME="son">
<FUNCTION NAME="light">
<FUNCTION NAME="temp">
<FUNCTION NAME="mic">
<FUNCTION NAME="accelX">
<FUNCTION NAME="accelY">
<FUNCTION NAME="magX">
<FUNCTION NAME="magY">
```

Figure 4.11. The mica sensor board library file.

were inline in the description. Figure 4.11 shows the contents of the mica sensor board library. The two SEARCH elements tell the toolchain where to find the TinyOS components on which the functions depend.

4.6.3 Toolchain

Compiling an ASVM generates several Java classes, which the ASVM toolchain uses to compile and install ASVM scripts. First, using `ncg`, the Makefile generates a Java class which has all of the externally relevant ASVM constants as public variables. The ASVM toolchain accesses this class using Java reflection to look up identifiers. For example, when a user writes a script for the reboot handler, the toolchain uses the constants class to look up the handler's identifier when it generates a new capsule.

The ASVM toolchain also depends on a family of classes representing TinyOS messages. The details of these classes depend on ASVM configuration parameters (e.g., the size of a capsule), so the toolchain generates them for each ASVM.

4.6.4 Example Scripts: Bombilla

To give a better sense of how the toolchain builds ASVMs, compiles scripts and installs capsules, we give a specific example.

Bombilla is a stock TinyScript ASVM that includes several handlers as well as general communication

```
<VM NAME="BombillaMicaVM" DESC="A simple VM that includes a range of
triggering events and functions." DIR="../../../../apps/BombillaMica">
<SEARCH PATH="../opcodes">
<SEARCH PATH="../contexts">
<SEARCH PATH="../languages">

<LOAD FILE="../sensorboards/micasb.vmsf">
<LANGUAGE NAME="tinyscript">

<FUNCTION NAME="bclear">
<FUNCTION NAME="bfull">
<FUNCTION NAME="bsize">
<FUNCTION NAME="bufsorta">
<FUNCTION NAME="bufsortd">
<FUNCTION NAME="eqtype">
<FUNCTION NAME="err">
<FUNCTION NAME="id">
<FUNCTION NAME="int">
<FUNCTION NAME="led">
<FUNCTION NAME="rand">
<FUNCTION NAME="send">
<FUNCTION NAME="sleep">
<FUNCTION NAME="uart">

<CONTEXT NAME="Trigger">
<CONTEXT NAME="Timer0">
<CONTEXT NAME="Timer1">
<CONTEXT NAME="Once">
<CONTEXT NAME="Reboot">
<CONTEXT NAME="Broadcast">
```

Figure 4.12. Bombilla description file.

Script		Assembly	Binary
private i;			
private j;			
buffer b;			
		bpush3 0	0x2b
uart(b);		uart	0xb6
		pushc6 1	0x54
for i=1 to 5000		setlocal3 0	0x21
	label0:	bpush3 0	0x2b
bclear(b);		bclear	0xa9
		pushc6 0	0x53
for j=0 step 0 until bfull(b)		setlocal3 1	0x22
	label1:	light	0xa2
		bpush3 0	0x2b
		add	0x02
b[] = light();		pop	0x97
		bpush3 0	0x2b
		bfull	0xaa
		not	0x08
next j		2jumps10 label1	0x15 0x08
		bpush3 0	0x2b
bsorta(b)		bufsorta	0xac
		getlocal3 0	0x19
		pushc6 1	0x54
		add	0x02
		copy	0x9e
		setlocal3 0	0x21
		3pushc16 5000	0x14 0x13 0x88
		lt	0x9b
next i		2jumpss10 label0	0x15
		bpush3 0	0x2b
uart(b);		uart	0xb6
		halt	0x00

Figure 4.13. Bombilla script that loops 500 times, filling a buffer with light readings and sorting the buffer. The program is 32 bytes long.

and sensing primitives. Figure 4.12 contains the description file of its mica variant (Telos versions include the Telos sensors). It supports the mica sensor board (Figure 4.11), tree-based aggregation (the `send` function), and simple broadcasts (the `bcast` function, included with the Broadcast handler).

Running the VMBuilder tool generates the four files. Compiling Bombilla for the mica2 platform, it uses 49kB of program ROM and 3416B of RAM. It uses 188 of the 256 available opcode values (there is space for an additional 68 functions).

Figure 4.13 shows a sample Bombilla script; the user writes the program with the TinyScript toolchain. The TinyScript compiler reads in the Bombilla metadata file to determine which functions the ASVM supports. The TinyScript toolchain compiles the script to assembly, shown in the center, which it passes to the ASVM assembler to convert into the bytecodes, shown on the right.

The ASVM toolchain places the binary into a code capsule, with the appropriate handler ID and version number. It breaks the capsule up into TinyOS capsule fragment packets, which it sends to a mote over the specified communication channel (e.g., serial port, TCP socket to an Ethernet adapter).

4.7 Evaluation

In this section, we evaluate the components of the ASVM architecture, measuring how each component contributes to a system's energy budget. We compare ASVMs against native programs and hand written runtime systems, measuring the overhead an ASVM imposes over other approaches.

4.7.1 Efficiency

For ASVMs to be a feasible mechanism for in-situ reprogramming, they cannot impose a significant energy overhead. If a user must decide between doubling a network lifetime and using an ASVM, the former will almost always win. Therefore, ASVMs must be efficient: they cannot impose a significant energy cost. To determine whether ASVMs satisfy this requirement, we evaluate each sub-component of the system.

Overhead Breakdown

As discussed in Section 3.3, ASVMs spend energy in two ways: propagation energy, spent by the radio, and execution energy, spent by the CPU. Propagation energy can be estimated as $P(n) = k + (p \cdot n)$, where n is the size of a program. Execution energy can be estimated as $E(N, T) = C(N) + I(N, T)$, where $C(n)$ is the cost of translating the network encoding into the on-mote representation, and $I(n, t)$ is the cost of execution over a time t. The next chapter addresses the propagation energy; here we evaluate $E(N, T)$.

Operation	Cycles	Time (μs)
Lock	32	8
Unlock	39	10
Run	1077	269
Analysis	15158	3790

Table 4.1. Cycle counts (on a mica class mote) for synchronization operations. The costs do not include the overhead of a function call (8 cycles). Depending on what and when the compiler inlines, this overhead may or may not be present.

Execution Energy: Interpretation

Our first evaluation of ASVM efficiency is a series of microbenchmarks of the scheduler. We compare ASVMs to the original version of Maté, which was a hand-tuned and monolithic VM.

Following the methodology we used in ASPLOS 2002 [64], we measured the bytecode interpretation overhead an ASVM imposes by writing a tight loop and counting how many times it ran in five seconds on a mica mote. The loop accessed a shared variable (which required lock checks through the concurrency manager). An ASVM can issue just under ten thousand instructions per second on a 4MHz mica, i.e., roughly 400 cycles per instruction. The ASVM decomposition imposes a 6% overhead over a similar loop in Maté, in exchange for handler and instruction set flexibility as well as race-free, deadlock-free parallelism.

We have not optimized the interpreter for CPU efficiency. The fact that high-level operations dominate program execution [64], combined with the fact that CPUs in sensor networks are generally idle, makes this overhead acceptable, although decreasing it with future work is of course desirable. For example, a KNearReduce function in the RegionsVM sends just under forty packets, and its ASVM scripting overhead is approximately 600 CPU cycles, the energy overhead is less than 0.03%. However, a cost of 400 cycles per bytecode means that implementing complex mathematical codes in an ASVM is inefficient; if an application domain needs significant processing, it should include appropriate operations.

Operation	Cycles
Type Check	7
Stack Overflow	29
Stack Underflow	28
Lock Held	19
Jump Bounds	20

Table 4.2. Cycle counts (on a mica class mote) for common safety checks. The costs do not include the overhead of a function call (8 cycles). Depending on what and when the compiler inlines, this overhead may or may not be present.

Execution Energy: Synchronization

We measured the overhead of ASVM concurrency control, using the cycle counter of a mica mote. Table 4.1 summarizes the results. Lock and unlock are acquiring or releasing a shared resource. Run is moving a thread to the run queue, obtaining all of its resources. Analysis is a full handler analysis. All values are averaged over 50 samples. These measurements were on an ASVM with 24 shared resources and a 128 byte handler. Locking and unlocking resources take on the order of a few microseconds, while a full program analysis for shared resource usage takes under a millisecond, approximately the energy cost of transmitting four bits.

These operations enable the concurrency manager to provide race-free and deadlock-free handler parallelism at a very low cost. By using implicit concurrency management, an ASVM can prevent many race condition bugs while keeping programs short and simple.

Execution Energy: Safety

An ASVM uses a wide range of runtime checks to ensure that bugs in ASVM programs do not crash a node. These checks also catch many common bugs in operation and handler implementations: for example, if a function implementation accidentally pushes a value onto the operand stack even though the function does not have a return value, then eventually the operand stack will overflow.

Table4.2 contains the cycle counts of five common safety checks. *Type check* is the cost of checking whether a variable is a particular ASVM type. This is a common check that may operations use. The

ASVM performs *stack overflow* and *stack underflow* checks whenever it pushes or pops the operand stack. They are more expensive than type checks because they require traversing a level of indirection in thread data structure. Whenever a thread accesses a shared resource, the *lock held* check ensures that it holds the resource: this check is necessary as the thread may have incorrectly released the lock. Finally, *jump bounds* checks that a jump instruction does not reference an instruction beyond the size of the current code image: a failed jump bound check indicates either program corruption or a script compilation bug.

All of these cycle counts are deterministic, and represent the common case (the check passes). Failing a check has a higher cost, as the operation must trigger an error condition in the VM engine.

Overhead Summary

Embedded microcontrollers are not high-performance computing devices. Unlike current PC processors, whose performance greatly depends on cache behavior or speculation mechanisms such as branch estimation and prefetching, microcontroller execution time is much more deterministic and mostly dependent on the instruction themselves. Interpretation provides a sandboxed code execution environment, but this safety comes at a significant per-instruction overhead.

Recall, however, that this overhead is effectively independent of the complexity of the code underlying an instruction. High level operations, such as sending a packet, can constitute thousands of CPU cycles: the few hundred cycles of interpretation overhead are small in comparison. Additionally, as programs rely heavily on application-specific functions – which often encapsulate complex logic – most CPU cycles are spent in application behavior rather than interpretation.

Understanding the roles each of these variables have in a full system requires a full system evaluation: we defer such an evaluation to later in this Chapter, in Section 4.7.3.

4.7.2 Conciseness

The energy equation $E(N,T) = C(N) + I(N,T)$ means that reducing code size improves both execution and propagation energy. The efficiency results in the previous subsection depend on the fact that ASVM

	Native	**RegionsVM**
Code (Flash)	19kB	39kB
Data (RAM)	2775B	3017B
Transmitted Program	19kB	71B

Table 4.3. Space utilization of native and RegionsVM regions implementations (bytes).

programs are very concise. As each instruction imposes an approximately constant overhead, reducing the number of instructions interpreted reduces overhead. Although there is a general tendency for concise programs to be more efficient, there are clear exceptions: a tight loop that executes a million times can be very short but very expensive.

Propagation energy ($C(N)$), however, is independent of the semantics of a program. Concise programs reduce both propagation energy and the memory required to store program images: this latter point can be a significant concern in RAM-limited mote designs.

In this subsection, we evaluate how the flexible ASVM boundary affects code conciseness. We evaluate conciseness by considering three sample programs from three different ASVMs. [2]

Regions Program

RegionsVM, designed for vehicle tracking, presents the abstract regions programming abstraction of MPI-like reductions over shared tuple spaces. Users write programs in TinyScript, and RegionsVM includes ASVM functions for the basic regions library; we obtained the regions source code from its authors. Figure 4.14 shows regions pseudocode proposed by Welsh at al. [111] next to actual TinyScript code that is functionally identical (it invokes all of the same library functions). The nesC components that present the regions library as ASVM functions are approximately 400 lines of nesC code.

In the native regions implementation, reprogramming a network requires installing an entire TinyOS image: the script in Figure 4.14(a) compiles to 19kB of binary code. In contrast, the RegionsVM script in Figure 4.14(b) compiles to 71 bytes of virtual code. Table 4.3 contains a decomposition of the resource utilization of the two implementations.

[2] We present the ASVMs themselves in greater depth in the next Chapter.

```
location = get_location();
/* Get 8 nearest neighbors */
region = k_nearest_region_create(8);

while(true) {
  reading = get_sensor_reading();

  /* Store local data as shared variables */
  region.putvar(reading_key, reading);
  region.putvar(reg_x_key, reading * location.x);
  region.putvar(reg_y_key, reading * location.y);

  if (reading > threshold) {
    /* ID of the node with the max value */
    max_id = region.reduce(OP_MAXID, reading_key);

    /* If I am the leader node... */
    if (max_id == my_id) {
      sum = region.reduce(OP_SUM, reading_key);
      sum_x = region.reduce(OP_SUM, reg_x_key);
      sum_y = region.reduce(OP_SUM, reg_y_key);
      centroid.x = sum_x / sum;
      centroid.y = sum_y / sum;
      send_to_basestation(centroid);
    }
  }
  sleep(periodic_delay);
}
```

(a) Regions Pseudocode

```
!! Create nearest neighbor region
KNearCreate();

for i = 1 until 0
  reading = int(mag());

  !! Store local data as shared variables
  KNearPutVar(0, reading);
  KNearPutVar(1, reading * LocX());
  KNearPutVar(2, reading * LocY());

  if (reading > threshold) then
    !! ID of the node with the max value
    max_id = KNearReduceMaxID(0);

    !! If I am the leader node
    if (max_id = my_id) then
      sum = KNearReduceAdd(0);
      sum_x = KNearReduceAdd(1);
      sum_y = KNearReduceAdd(2);
      buffer[0] = sum_x / sum;
      buffer[1] = sum_y / sum;
      send(buffer);
    end if
  end if
  sleep(periodic_delay);
next i
```

(b) TinyScript Code

Figure 4.14. Regions Pseudocode and Corresponding TinyScript. The pseudocode is from "Programming Sensor Networks Using Abstract Regions." The TinyScript program on the right compiles to 71 bytes of binary code.

Name	TinySQL
Simple	`SELECT id,parent,temp INTERVAL 50s`
Conditional	`SELECT id, expdecay(humidity, 3) WHERE parent > 0 INTERVAL 50s`
SpatialAvg	`SELECT AVG(temp) INTERVAL 50s`

Table 4.4. The three queries used to QueryVM versus TinyDB. TinyDB does not directly support time-based aggregates such as `expdecay`, so in TinyDB we omit the aggregate.

	Size (bytes)		Increase
Query	TinyDB	QueryVM	
Simple	93	105	12%
Conditional	124	167	34%
SpatialAvg	62	127	104%

Table 4.5. Query size in TinyDB and QueryVM. The queries refer to those in Table 4.4.

TinySQL Queries

The comparison between RegionsVM and the native regions implementation showed that application-level program representations can be orders of magnitude smaller than TinyOS binaries. This still leaves the question of whether ASVMs are a concise application-level representation. This is difficult to evaluate, as conciseness and flexibility are an inherent tradeoff: the best representation for a specific instance of an application domain is a single bytecode: "run program." Therefore, rather than pure conciseness, a more accurate measure is how concise programs are with a given degree of flexibility.

To evaluate how ASVMs perform in this regard, we built an ASVM for TinySQL queries, named QueryVM. TinyDB is a customized and hand-built system for executing TinySQL queries. QueryVM is more flexible that TinyDB, as it allows users to write new aggregates in motlle code, but the comparison is reasonably close. Comparing the size of a TinyDB query and a QueryVM query is a good measure of how concisely an ASVM can encode a program. We wrote three representative queries, shown in Table 4.4 and compared their sizes in TinyDB and QueryVM. We use these same queries in Section 4.7.3 to evaluate QueryVM's energy efficiency. Table 4.5 contains the results.

QueryVM's programs are 12–104% larger than TinyDB's. In the worst case, this represents propagating a query requiring six instead of three packets.

4.7.3 Whole System Evaluation

The previous sections used a series of microbenchmarks to evaluate the overhead and cost of various ASVM abstractions, services, and functionality. This section evaluates ASVMs as whole systems. The principal metric for sensor networks is energy. Using QueryVM, we compare the energy consumption of an ASVM network versus TinyDB and versus hand-written query programs.

The previous sections measured the cost of each abstraction and service that an ASVM provides. This section evaluates ASVMs as a complete system. Using three representative TinySQL queries, we compare QueryVM against TinyDB. We quantify the overhead QueryVM imposes over hand-written nesC implementations of the queries.

TinySQL, TinyDB, and QueryVM

TinySQL is a language that presents a sensor network as a streaming database. It both restricts and expands upon SQL. For example, TinySQL does not support joins, but it introduces the notion of a "sample period" at which the query repeats. TinySQL supports both simple data collection and aggregate queries such as

SELECT AVG(temperature) INTERVAL 50s

to measure the average temperature of the network. The latter allows in-network processing to reduce the amount of traffic sent, by aggregating as nodes route data up a collection tree [71].

TinyDB is a TinyOS system that executes TinySQL queries. In the TinyDB model, a user writes a TinySQL query, which the TinyDB toolchain compiles into a binary form and installs on a mote TinyDB base station. TinyDB floods the query through the network. TinyDB executes streaming queries by breaking time into *epochs*. Each epoch represents a time interval of the desired sampling frequency. Epochs allow TinyDB to know which sensor readings are part of the same result, which is necessary when aggregating.

To evaluate whether ASVMs are flexible enough to efficiently support TinySQL, we build QueryVM. QueryVM is a motlle-based ASVM designed to support the execution of TinySQL data collection queries.

```
// Initialise the operator
expdecay_make = fn (bits) vector(bits, 0);
// Update the operator (s is result from make)
expdecay_get = fn (s, val)
  // Update and return the average (s[0] is BITS)
  s[1] = s[1] - (s[1] >> s[0]) + (attr() >> s[0]);
```

Figure 4.15. An exponentially decaying average operator for TinySQL, in motlle.

Our TinySQL compiler generates motlle code from queries such as those shown in Table 4.4. The generated code is responsible for timing, message layout and aggregating data on each hop up the routing tree. The code in Figure 4.9 is essentially the same as that generated for the Simple query. Users can write new attributes or operators for TinySQL using snippets of motlle code. For instance, Figure 4.15 shows two lines of motlle code to add an exponentially-decaying average operator, which an example in Table 4.4 uses.

QueryVM includes functions and handlers to support multi-hop communication, epoch handling and aggregation. QueryVM programs use the same tree based collection layer, MintRoute [114], that TinyDB uses. QueryVM includes epoch-handling primitives to avoid replicating epoch-handling logic in every program (see usage in Figure 4.9). Temporal or spatial (across nodes) averaging logic can readily be expressed in motlle, but including common aggregates in QueryVM reduces program size and increases execution efficiency.

Three Queries

We compare QueryVM with TinyDB using the three queries in Table 4.4. The first query, Simple, is basic data collection: every node reports three data values. The Simple query uses no local filtering or in-network aggregation. The Conditional query uses a local exponentially weighted moving average data filter, and only reports the filtered value if a local predicate (the node's parent) is met. Each node's data values in the Conditional query are independent: there is no in-network aggregation. Finally, the SpatialAvg query computes the whole-network average using in-network aggregation. For each epoch, a node combines all of

Figure 4.16. Tree topology used in QueryVM/TinyDB experiments. The square node is the tree root.

its children's averages with its own before reporting. The result is a single result from the entire network, rather than a result from each node.

Whole Network

We ran the queries on a network of 40 mica2 motes spread across the ceiling of an office building. Motes had the mts400 weather board from Crossbow Technologies. Environmental changes can dynamically alter

	Size (bytes)		Energy (mW)		Yield	
Query	TinyDB	VM	TinyDB	VM	TinyDB	VM
Simple	93	105	5.6	4.5	73%	74%
Conditional	124	167	4.2	4.0	65%	79%
SpatialAvg	62	127	3.3	3.1	46%	55%

Table 4.6. Query size, power consumption and yield in TinyDB and QueryVM. Yield is the percentage of expected results received.

ad-hoc routing trees (e.g., choosing a 98% link over a 96% link), changing the forwarding pattern and greatly affecting energy consumption. As we are unable to control the wireless emissions in real world settings, these sorts of changes make perfect experimental repeatability very difficult, if not impossible. Therefore, we used a static, stable tree in our experiments, to provide an even basis for comparison across the implementations. We obtained this tree by running the routing algorithm for a few hours, extracting the parent sets, then explicitly setting node parents to this topology, shown in Figure 4.16. Experiments run on adaptive trees were consistent with the results presented below. We measured the power consumption of a mote with a single child, physically close to the root of the multi-hop network. Its power reflects a mote that overhears a lot of traffic but which sends relatively few messages (a common case). In each of the queries, a node sends a data packet every 50 seconds, and the routing protocol sends a route update packet every two epochs (100 seconds). We measured the average power draw of the instrumented node over 16 intervals of 100 seconds, sampling at 100 Hz (10,000 instantaneous samples).

Table 4.6 presents the results from these experiments. For the three sample queries, QueryVM consumes 5% to 20% less energy than TinyDB. However, we do not believe all of this improvement to be fundamental to the two approaches. The differences in yield mean that the measured mote is overhearing different numbers of messages — this increases QueryVM's power draw. Conversely, having larger packets increases TinyDB's power draw — based on the 325μJ per-packet cost, we estimate a cost of 0.2–0.5mW depending on the query and how well the measured mote hears more distant motes.

However, these are not the only factors at work, as shown by experiments with a native TinyOS implementation of the three queries. We ran these native implementations in two scenarios. In the first scenario, we booted all of the nodes at the same time, so their operation was closely synchronized. In the second, we staggered node boots over the fifty second sampling interval. Figure 4.17 shows the power draw for these two scenarios, alongside that of TinyDB and QueryVM. In the synchronized case, yields for the native implementations varied between 65% and 74%, in the staggered case, yields were between 90% and 97%. As these results show, details of the timing of transmissions have major effects on yield and power consumption. To separate these networking effects from basic system performance, we repeated the experiments in a two node network.

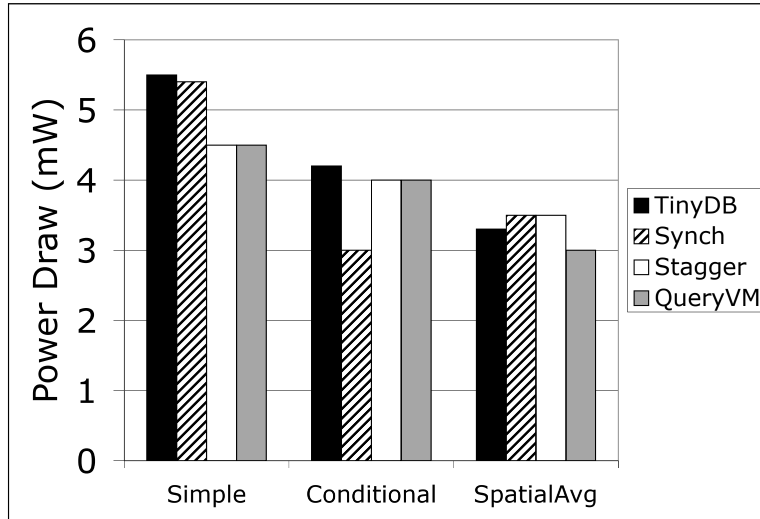


Figure 4.17. Power consumption of TinyDB, QueryVM, and nesC implementations. Synch is the nesC implementation when nodes start at the same time. Stagger is when the nodes start times are staggered.

Single Node

In our two-node experiments, the measured mote executes the query and sends results, and the second mote is a passive base station. As the measured node does not forward any packets or contend with other transmitters, its energy consumption is the cost of query execution and reporting. The extra cost of sending TinyDB's larger result packets is negligible (.01mW extra average power draw). We ran these experiments longer than the full network ones: rather than 16 intervals of length 100 seconds (25 minutes), we measured for 128 intervals (3.5 hours).

The results, presented in Figure 4.18, show that QueryVM has a 5–20% energy performance improvement over TinyDB. Even though an ASVM based on reusable software components and a common template, rather than a hand-coded, vertically integrated system, QueryVM imposes less of an energy burden on a deployment. In practice, power draw in a real network is dominated by networking costs — QueryVM's 0.25mW advantage in Figure 4.18 would give at most 8% longer lifetime based on the power draws of Figure 4.17.

To determine where QueryVM's power goes, we compared it to four hand coded TinyOS programs. The first program did not process a query: it just listened for messages and handled system timers. This

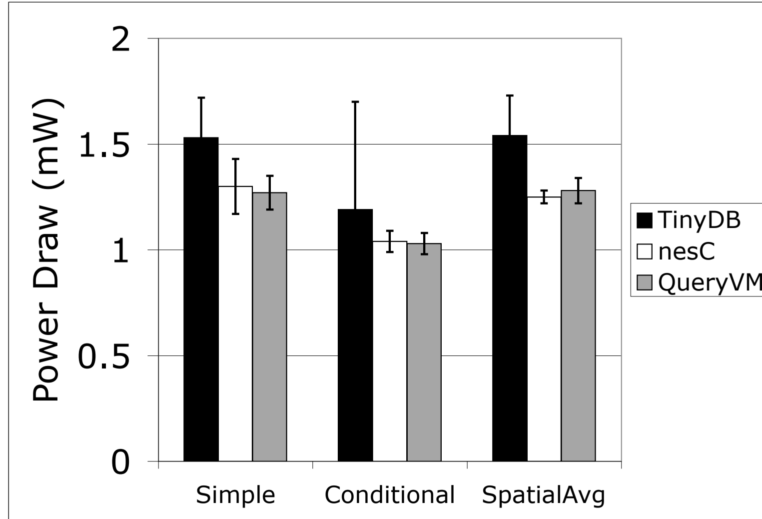


Figure 4.18. Average power draw measurements in a two node network. For the Conditional query, the monitored node has parent = 0, therefore it sends no packets. The error bars are the standard deviation of the per-interval samples.

allows us to distinguish the cost of executing a query from the underlying cost of the system. The other three were the nesC implementations of the queries used for Figure 4.17. They allow us to distinguish the cost of executing a query itself from the overhead an ASVM runtime imposes. The basic system cost was 0.76 mW. Figure 4.18 shows the comparison between QueryVM and a hand-coded nesC implementation of the query. The queries cost 0.28–0.54 mW, and the cost of the ASVM is negligible.

This negligible cost is not surprising: for instance, for the conditional query, QueryVM executes 49 instructions per sample period, which will consume approximately 5ms of CPU time. Even on a mica2 node, whose CPU power draw is a whopping 33 mW due to an external oscillator (other platforms draw 3–8 mW), this corresponds to an average power cost of 3.3μW. In the 40 node network, the cost of overhearing on other node's results will increase power draw by another 20μW. Finally, QueryVM sends viral code maintenance messages every 100 minutes (in steady state), corresponding to an average power draw of 1.6μW.

From the results in Table 4.6, with a power consumption of 4.5mW, a pair of AA batteries (2700mAh, of which approximately two thirds is usable by a mote) would last for 50 days. By lowering the sample rate (every fifty seconds is a reasonably high rate) and other optimizations, we believe that lifetimes of three

	None	Operation	Script
Iteration	16.0	16.4	62.2
Sort Time	-	0.4	46.2

Table 4.7. Execution time of three scripts, in milliseconds. None is the version that did not sort, operation is the version that used an operation, while script is the version that sorted in script code.

months or more are readily achievable. Additionally, the energy cost of ASVM interpretation is a negligible portion of the whole system energy budget. This suggests that ASVM-based active sensor networking can be a realistic option for long term, low-duty-cycle data collection deployments.

4.7.4 Cost/Benefit Tradeoff

Building an ASVM that encapsulates high-level application abstractions in functions can lead to very concise code, thereby reducing energy costs. However, this benefit comes at the cost of flexibility. While a programmer can write additional logic on top of the ASVM functions and handlers, the native/virtual boundary prevents writing logic below them. The basic tradeoff in defining the ASVM boundary is between flexibility and efficiency.

To obtain some insight into the tradeoff between including functions and writing operations in script code, we wrote three scripts. The first script is a loop that fills an array with sensor readings. The second script fills the array with sensor readings and sorts the array with an operation (`bufsorta`, which is an insertion sort). The third script also insertion sorts the array, but does so in TinyScript, rather than using an operation. To measure the execution time of each script, we placed it in a 5000 iteration loop and sent a UART packet at script start and end. Table 4.7 shows the results. Sorting the array with script code takes 115 times as an ASVM sort function, and dominates script execution time. Interpretation is inefficient, but pushing common and expensive operations into native code with functions minimizes the amount of interpretation.

4.7.5 Analysis

Section 4.7.3 showed that an ASVM is an effective way to efficiently provide a high-level programming abstraction to users. It is by no means the only way, however. There are two other obvious approaches: using a standard virtual machine, such as Java, and sending very lightweight native programs.

As a language, Java may be a suitable way to program a sensor network, although we believe a very efficient implementation might require simplifying or removing some features, such as reflection. Java Card has taken such an approach, essentially designing an ASVM for smart cards that supports a limited subset of Java and different program file formats. Although Java Card supports a single application domain, it does provide guidance on how an ASVM could support a Java-like language.

Native code is another possible solution: instead of being bytecode-based, programs could be native code stringing together a series of library calls. As sensor mote CPUs are usually idle, the benefit native code provides — more efficient CPU utilization — is minimal, unless a user wants to write complex mathematical codes. In the ASVM model, these codes should be written in nesC, and exposed to scripts as functions. Additionally, native code poses many complexities and difficulties, which greatly outweigh this minimal benefit, including safety, conciseness, and platform dependence. However, the SOS operating system suggests ways in which ASVMs could support dynamic addition of new functions.

ASVMs share the same high-level goal as active networking: dynamic control of in-network processing. The sort of processing proposed by systems such as ANTS and PLANet, however, is very different than that which we see in sensor nets. Although routing nodes in an active Internet can process data, edge systems are still predominantly responsible for generating that data. Correspondingly, much of active networking research focused on protocol deployment. In contrast, motes simultaneously play the role of both a router and a data generator. Instead of providing a service to edge applications, active sensor nodes are the application. Section 4.7.3 showed how an ASVM — QueryVM — can simultaneously support both SQL-like queries and motlle programs, compiling both to a shared instruction set. In addition to being more energy efficient than a similar TinyDB system, QueryVM is more flexible. Similarly, RegionsVM has several benefits —

code size, concurrency, and safety — over the native regions implementation. We believe these advantages are a direct result of how ASVMs decompose programming into three distinct layers.

TinyDB is a two-layer approach that combines the top two layers: its programs are binary encodings of an SQL query. This forces a mote to parse and interpret the query, and determine what actions to take on all of the different events coming into the system. It trades off flexibility and execution efficiency for propagation efficiency. Separating the programming layer and transmission layer, as QueryVM does, leads to greater program flexibility and more efficient execution.

Regions is a two-layer approach that combines the bottom two layers: its programs are TinyOS images. Using the TinyOS concurrency model, rather than a virtual one, limits the native regions implementation to a single thread. Additionally, even though its programs are only a few lines long — compiling to seventy bytes in RegionsVM — compiling to a TinyOS image makes its programs tens of kilobytes long, trading off propagation efficiency and safety for execution efficiency. Separating the transmission layer from the execution layer, as RegionsVM does, allows high-level abstractions to minimize execution overhead and provides safety.

4.8 Related Work

SOS is a sensor network operating system that supports dynamic native code updates through a loadable module system [46]. This allows small and incremental binary updates, but requires levels of function call indirection. SOS therefore between the extremes of TinyOS and ASVMs. Its propagation cost is less than TinyOS and greater than ASVMs, while its execution overhead is greater than TinyOS but less than ASVMs. By using native code to achieve this middle ground, SOS cannot provide all of the safety guarantees that an ASVM can. Still, the SOS approach suggests ways in which ASVMs could dynamically install new functions.

The Impala middleware system, like SOS, allows users to dynamically install native code modules [68]. However, unlike SOS, which allows modules to both call a kernel and invoke each other, Impala limits modules to the kernel interfaces. Like ASVMs, these interfaces are event driven, and bear a degree of similarity

to early versions of Maté [64]. Unlike ASVMs, however, Impala does not provide general mechanisms to change its triggering events, as it is designed for a particular application domain, ZebraNet [57].

SensorWare [15] is another proposal for programming nodes using an interpreter: it proposes using Tcl scripts. For the devices SensorWare is designed for — iPAQs with megabytes of RAM — the verbose program representation and on-node Tcl interpreter can be acceptable overheads. On a mote, however, they are not.

Customizable and extensible abstraction boundaries, such as those ASVMs provide, have a long history in operating systems research. Systems such as scheduler activations [5] show that allowing applications to cooperate with a runtime through rich boundaries can greatly improve application performance. Operating systems such as exokernel [58] and SPIN [10] take a more aggressive approach, allowing users to write the interface and improve performance through increased control. In sensor networks, performance — the general goal of *more*, whether it be bandwidth, or operations per second — is rarely a primary metric, as low duty cycles make resources plentiful. Instead, robustness and energy efficiency are the important metrics. Code interpretation has two common benefits besides protection: platform independence and conciseness.

While the commercial OS market has led to platform independence being a contentious topic, it has a long history in computer science research, particularly hardware architecture. In the architecture domain, the issue faced was the need to support old codes on evolving instruction sets. For example, when Digital launched its Alpha series of processors, one requirement was to be able to run existing VAX and MIPS software on it [98]. In this case, the motivation was backwards compatibility, rather than future interoperability, as it is with languages such as Java [44]. Although code is stored in a platform independent form, it can be readily translated into native instructions: the growing gap between disk bandwidth and processing speeds means that the translation overhead can be small even for programs that run for only a short time [3].

The comparative costs of CPU cycles and disk operations also means that storing more concise codes on disk — which must be translated to more CPU-efficient codes — can reduce overall execution times [34]. In systems such as Java, download time usually dwarfs execution time. Being able to download a smaller program that requires translation would be a perceived performance boost. Code conciseness improves

performance when transfer times are the bottleneck. It is also needed when memory is a bottleneck, trading increased time for reduced space [38].

As discussed in Section 3.6, there has been a lot of work in minimizing Java VMs to be able to run on embedded devices. These efforts, however, are subject to the same limitations as any other hard boundary: limited efficiency across a wide range of domains.

Chapter 5

Code Propagation

One of the core services the Maté template provides is automatic code propagation. As mentioned in Section 5.5, a Maté ASVM uses the Trickle algorithm to advertise three kinds of data. This section motivates and describes the algorithm.

The radio is the principal energy consumer of a sensor network mote. Communication determines system lifetime, so an effective code propagation algorithm must send few packets. However, a network can be in a useless state while code is propagating, as multiple programs may be running concurrently. Transition time is wasted time, and wasted time is wasted energy. An effective code propagation algorithm must also propagate new code quickly. As code size influences propagation time, minimizing code size improves propagation rates.

Wireless sensor networks may operate at a scale of hundreds, thousands, or more. Unlike Internet based systems, which represent a wide range of devices linked through a common network protocol, sensor networks are independent, application specific deployments. They exhibit highly transient loss patterns that are susceptible to changes in environmental conditions [103]. Asymmetric links are common, and prior work has shown network behavior to often be worse indoors than out, predominantly due to multi-path effects [115]. Motes come and go, due to temporary disconnections, failure, and network repopulation.

As new code must eventually propagate to every mote in a network, but network membership is not static, propagation must be a continuous effort.

Propagating code is costly; learning *when* to propagate code is even more so. Motes must periodically communicate to learn when there is new code. To reduce energy costs, motes can transmit metadata to determine when code is needed. Even for binary images, this periodic metadata exchange overwhelms the cost of transmitting code when it is needed. Sending a full TinyDB binary image (≈ 64 KB) costs approximately the same as transmitting a forty byte metadata summary once a minute for a day. In Maté, TinyDB, and similar systems, this tradeoff is even more pronounced: sending a few metadata packets costs the same as sending an entire program. The communication to learn when code is needed overwhelms the cost of actually propagating that code.

The first step towards efficient code propagation is efficient algorithm for determining when motes should propagate code, which can be used to trigger the actual code transfer. Such an algorithm has three needed properties:

Low Maintenance: When a network is in a stable state, metadata exchanges should be infrequent, just enough to ensure that the network has a single program. The transmission rate should be configurable to meet an application energy budget; this can vary from transmitting once a minute to every few hours.

Rapid Propagation: When the network discovers motes that need updates, code must propagate rapidly. Propagation should not take more than a minute or two more than the time required for transmission, even for large networks that are tens of hops across. Code must eventually propagate to every mote.

Scalability: The protocol must maintain its other properties in wide ranges of network density, from motes having a few to hundreds of network neighbors. It cannot require *a priori* density information, as density will change due to environmental effects and node failure.

This section describes the Maté ASVM code propagation protocol. The protocol is built on top of an algorithm called Trickle, developed for this purpose, which this chapter presents and evaluates. Trickle is

also used in several systems besides Maté, such as the binary dissemination service Deluge [52], and the SNMS MIB interface Drip [107].

Borrowing techniques from the epidemic, scalable multicast, and wireless broadcast literatures, Trickle regulates itself using a local "polite gossip" to exchange code metadata (we present a detailed discussion of Trickle with regards to this prior work to Section 5.8). Each mote periodically broadcasts metadata describing what code it has. However, if a mote hears gossip about identical metadata to its own, it stays quiet. When a mote hears old gossip, it triggers a code update, so the gossiper can be brought up to date. To achieve both rapid propagation and a low maintenance overhead, motes adjust the length of their gossiping attention spans, communicating more often when there is new code. Trickle can also be used as a general wireless suppression algorithm; the Maté storage stack uses a hierarchy of network trickles to propagate code.

Trickle meets the three requirements. It imposes a maintenance overhead on the order of a few packets an hour (which can easily be pushed lower), can be used to propagate updates across multi-hop networks in tens of seconds, and scales to thousand-fold changes in network density. In addition, it handles network repopulation, is robust to network transience, loss, and disconnection, and requires very little state (in the ASVM implementation, eleven bytes).

5.1 Trickle Algorithm

In this section, we describe the basic Trickle algorithm colloquially, as well as its conceptual basis. In Section 5.3, we describe the algorithm more formally, and discuss how it maintains that motes are up to date. In Section 5.4, we show Trickle can rapidly signal code propagation by dynamically adjusting time intervals.

5.1.1 Overview

Trickle's basic primitive is simple: every so often, a mote transmits code metadata if it has not heard a few other motes transmit the same thing. This allows Trickle to scale to thousand-fold variations in network density, quickly propagate updates, distribute transmission load evenly, be robust to transient disconnections, handle network repopulations, and impose a maintenance overhead on the order of a few packets per hour per mote.

Trickle sends all messages to the local broadcast address. There are two possible results to a Trickle broadcast: either every mote that hears the message is up to date, or a recipient detects the need for an update. Detection can be the result of either an out-of-date mote hearing someone has new code, or an updated mote hearing someone has old code. As long as every mote communicates somehow – either receives or transmits – the need for an update will be detected.

For example, if mote A broadcasts that it has code ϕ_x, but B has code ϕ_{x+1}, then B knows that A needs an update. Similarly, if B broadcasts that it has ϕ_{x+1}, A knows that it needs an update. If B starts sending updates, then all of its neighbors can receive them without having to advertise their need. Some of these recipients might not even have heard A's transmission.

In this example, it does not matter who first transmits, A or B; either case will detect the inconsistency. All that matters is that some motes communicate with one another at some nonzero rate; we will informally call this the "communication rate." As long as the network is connected and there is some minimum communication rate for each mote, everyone will stay up to date.

The fact that communication can be either transmission or reception enables Trickle to operate in sparse as well as dense networks. A single, disconnected mote must transmit at the communication rate. In a lossless, single-hop network of size n, the sum of transmissions over the network is the communication rate, so for each mote it is $\frac{1}{n}$. Sparser networks require more transmissions per mote, but utilization of the radio channel *over space* will not increase. This is an important property in wireless networks, where the channel is a valuable shared resource. Additionally, reducing transmissions in dense networks conserves system energy.

We begin in Section 5.2 by defining the experimental methodology we use to evaluate TOSSIM. Section 5.3 describes Trickle's maintenance algorithm, which tries to keep a constant communication rate. We analyze its performance (in terms of transmissions and communication) in the idealized case of a single-hop lossless network with perfect time synchronization. We relax each of these assumptions by introducing loss, removing synchronization, and using a multi-hop network. We show how each relaxation changes the behavior of Trickle, and, in the case of synchronization, modify the algorithm slightly to accommodate.

5.2 Methodology

We use three different platforms to investigate and evaluate Trickle. The first is a high-level, abstract algorithmic simulator written especially for this study. The second is TOSSIM [66], a bit-level mote simulator for TinyOS, a sensor network operating system [51]. TOSSIM compiles directly from TinyOS code. Finally, we used TinyOS mica-2 motes for empirical studies, to validate our simulation results and prove the real-world effectiveness of Trickle. The same implementation of Trickle ran on motes and in TOSSIM.

5.2.1 Abstract Simulation

To quickly evaluate Trickle under controlled conditions, we implemented a Trickle-specific algorithmic simulator. Little more than an event queue, it allows configuration of all of Trickle's parameters, run duration, the boot time of motes, and a uniform packet loss rate (same for all links) across a single hop network. Its output is a packet send count.

5.2.2 TOSSIM

The TOSSIM simulator compiles directly from TinyOS code, simulating complete programs from application level logic to the network at a bit level [66]. It simulates the implementation of the entire TinyOS network stack, including its CSMA protocol, data encodings, CRC checks, collisions, and packet timing. TOSSIM models mote connectivity as a directed graph, where vertices are motes and edges are links;

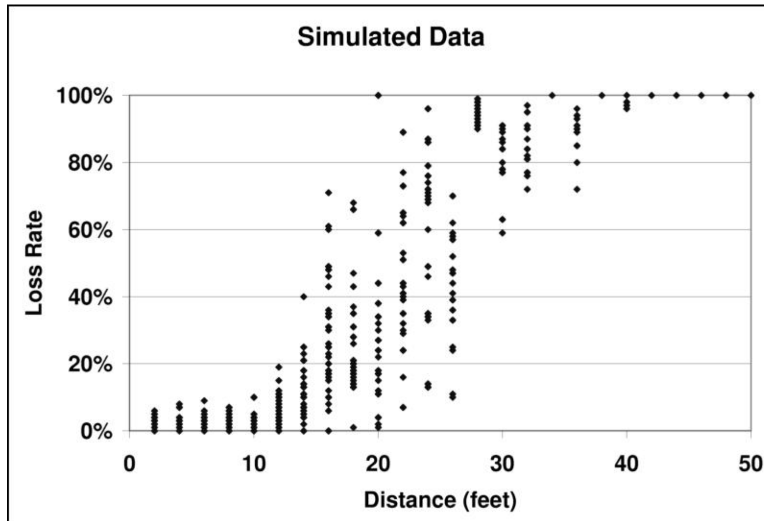


Figure 5.1. TOSSIM packet loss rates over distance

each link has a bit error rate, and as the graph is directed, link error rates can be asymmetric. This occurs when only one direction has good connectivity, a phenomenon that several empirical studies have observed [39, 115, 20]. The networking stack (based on the mica platform implementation) can handle approximately forty packets per second, with each carrying a 36 byte payload.

To generate network topologies, we used TOSSIM's empirical model, based on data gathered from TinyOS motes [39]. Figure 5.1 shows an experiment illustrating the model's packet loss rates over distance (in feet). As link directions are sampled independently, intermediate distances such as twenty feet commonly exhibit link asymmetry. Physical topologies are fed into the loss distribution function, producing a loss topology. In our studies, link error rates were constant for the duration of a simulation, but packet loss rates could be affected by dynamic interactions such as collisions at a receiver.

In addition to standard bit-level simulations, we used a modified version of TOSSIM that supports packet-level simulations. This version simulates loss due to packet corruption from bit errors, but does not model collisions. By comparing the results of the full bit-level simulation and this simpler packet-level simulation, we can ascertain when packet collisions – failures of the underlying MAC – are the cause of protocol behavior. We refer to the full TOSSIM simulation as TOSSIM-bit, and the packet level simulation as TOSSIM-packet.

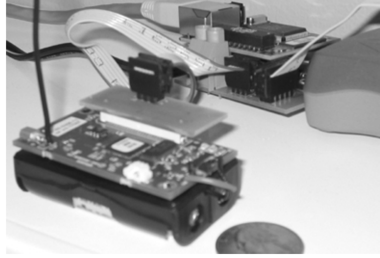


Figure 5.2. The TinyOS mica2

5.2.3 TinyOS motes

In our empirical code propagation experiments, we used TinyOS mica2 and mica2dot motes. These motes provide 128KB of program memory, 4KB of RAM, and a 7MHz 8-bit microcontroller for a processor. The radio transmits at 19.2 Kbit, which after encoding and media access, is approximately forty TinyOS packets/second, each with a thirty-six byte data payload. For propagation experiments, we instrumented mica2 motes with a special hardware device that bridges their UART to TCP; other computers can connect to the mote with a TCP socket to read and write data to the mote. We used this to obtain millisecond granularity timestamps on network events. Figure 5.2 shows a picture of one of the mica2 motes used in our experiments.

We performed three empirical studies. In the first, we evaluated Trickle's scalability by placing varying number of motes on a table, with the transmission strength set very low to create a small multi-hop network. In the second, we evaluated Trickle's signaling rate with a nineteen mote network in an office area, approximately 160' by 40'. Section 5.4 presents these experiments in greater depth. In final empirical study, we evaluated the ASVM propagation protocol with a 75 node testbed on the fourth floor of Soda Hall. Sections 5.5 presents the experimental setup in greater depth.

5.3 Maintenance

Trickle uses "polite gossip" to exchange code metadata with nearby network neighbors. It breaks time into intervals, and at a random point in each interval, it considers broadcasting its code metadata. If Trickle

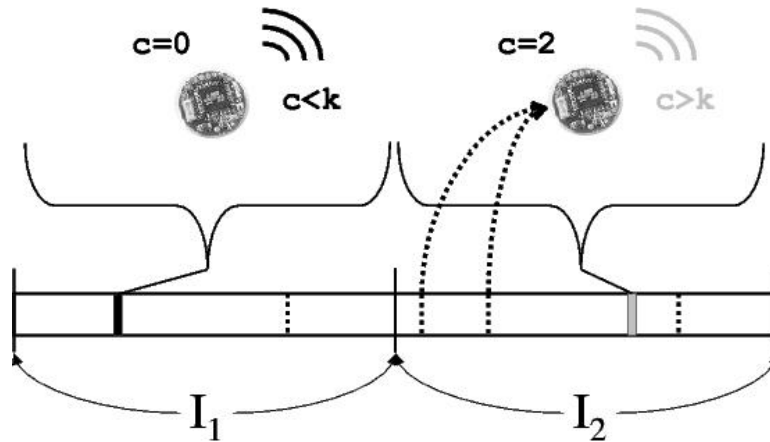


Figure 5.3. Trickle maintenance with a k of 1. Dark boxes are transmissions, gray boxes are suppressed transmissions, and dotted lines are heard transmissions. Solid lines mark interval boundaries. Both I_1 and I_2 are of length τ.

has already heard several other motes gossip the same metadata in this interval, it politely stays quiet: repeating what someone else has said is rude.

When a mote hears that a neighbor is behind the times (it hears older metadata), it brings everyone nearby up to date by broadcasting the needed pieces of code. When a mote hears that it is behind the times, it repeats the latest news it knows of (its own metadata); following the first rule, this triggers motes with newer code to broadcast it.

More formally, each mote maintains a counter c, a threshold k, and a timer t in the range $(0, \tau]$. k is a small, fixed integer (e.g., 1 or 2) and τ is a time constant. We discuss the selection of τ in depth in Section 5.4. When a mote hears metadata identical to its own, it increments c. At time t, the mote broadcasts its metadata if $c < k$. When the interval of size τ completes, c is reset to zero and t is reset to a new random value in the range $[0, \tau]$. If a mote with code ϕ_x hears a summary for ϕ_{x-y}, it broadcasts the code necessary to bring ϕ_{x-y} up to ϕ_x. If it hears a summary for ϕ_{x+y}, it broadcasts its own summary, triggering the mote with ϕ_{x+y} to send updates.

Figure 5.3 has a visualization of Trickle in operation on a single mote for two intervals of length τ with a k of 1 and no new code. In the first interval, I_1, the mote does not hear any transmissions before its t, and

broadcasts. In the second interval, I_2, it hears two broadcasts of metadata identical to its, and so suppresses its broadcast.

Using the Trickle algorithm, each mote broadcasts a summary of its data at most once per period τ. If a mote hears k motes with the same program before it transmits, it suppresses its own transmission. In perfect network conditions – a lossless, single-hop topology – there will be k transmissions every τ. If there are n motes and m non-interfering single-hop networks, there will be km transmissions, which is independent of n. Instead of fixing the per-mote send rate, Trickle dynamically regulates its send rate to the network density to meet a communication rate, requiring no a priori assumptions on the topology. In each interval τ, the sum of receptions and sends of each mote is k.

The random selection of t uniformly distributes the choice of who broadcasts in a given interval. This evenly spreads the transmission energy load across the network. If a mote with n neighbors needs an update, the expected latency to discover this from the beginning of the interval is $\frac{\tau}{n+1}$. Detection happens either because the mote transmits its summary, which will cause others to send updates, or because another mote transmits a newer summary. A large τ has a lower energy overhead (in terms of packet send rate), but also has a higher discovery latency. Conversely, a small τ sends more messages but discovers updates more quickly.

This km transmission count depends on three assumptions: no packet loss, perfect interval synchronization, and a single-hop network. We visit and then relax each of these assumptions in turn. Discussing each assumption separately allows us to examine the effect of each, and in the case of interval synchronization, helps us make a slight modification to restore scalability.

5.3.1 Maintenance with Loss

The above results assume that motes hear every transmission; in real-world sensor networks, this is rarely the case. Figure 5.4 shows how packet loss rates affect the number of Trickle transmissions per interval in a single-hop network as density increases. These results are from the abstract simulator, with

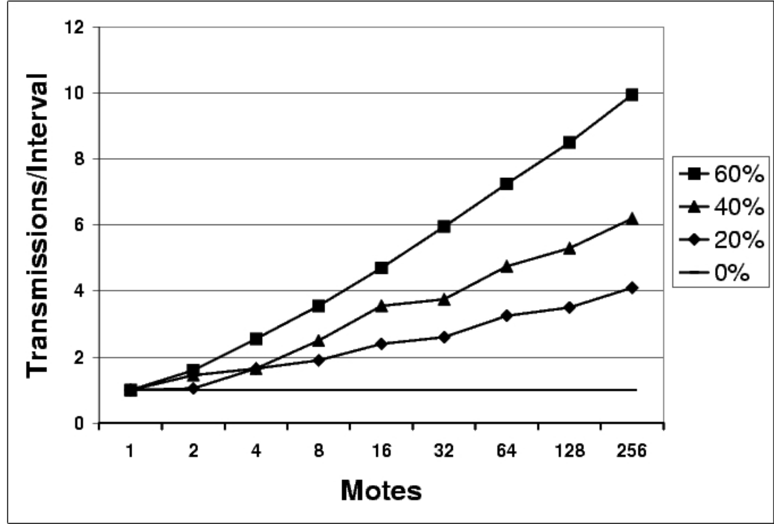


Figure 5.4. Number of transmissions as density increases for different packet loss rates.

$k = 1$. Each line is a uniform loss rate for all node pairs. For a given rate, the number of transmissions grows with density at $O(log(n))$.

This logarithmic behavior represents the probability that a single mote misses a number of transmissions. For example, with a 10% loss rate, there is a 10% chance a mote will miss a single packet. If a mote misses a packet, it will transmit, resulting in two transmissions. There is correspondingly a 1% chance it will miss two, leading to three transmissions, and a 0.1% chance it will miss three, leading to four. In the extreme case of a 100% loss rate, each mote is by itself: transmissions scale linearly.

Unfortunately, to maintain a per-interval minimum communication rate, this logarithmic scaling is inescapable: $O(log(n))$ is the best-case behavior. The increase in communication represents satisfying the requirements of the worst case mote; in order to do so, the expected case must transmit a little bit more. Some motes do not hear the gossip the first time someone says it, and need it repeated. In the rest of this work, we consider $O(log(n))$ to be the desired scalability.

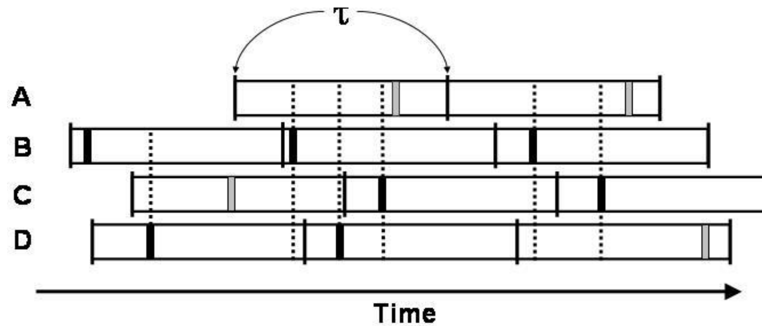


Figure 5.5. The short listen problem for motes A, B, C, and D. Dark bars represent transmissions, light bars suppressed transmissions, and dashed lines are receptions. Tick marks indicate interval boundaries. Mote B transmits in all three intervals.

5.3.2 Maintenance without Synchronization

The above results assume that all motes have synchronized intervals. Inevitably, time synchronization imposes a communication, and therefore energy, overhead. While some networks can provide time synchronization to Trickle, others cannot. Therefore, Trickle should be able to work in the absence of this primitive.

Unfortunately, without synchronization, Trickle can suffer from the *short-listen* problem. Some subset of motes gossip soon after the beginning of their interval, listening for only a short time, before anyone else has a chance to speak up. If all of the intervals are synchronized, the first gossip will quiet everyone else. However, if not synchronized, it might be that a mote's interval begins just after the broadcast, and it too has chosen a short listening period. This results in redundant transmissions.

Figure 5.5 shows an instance of this phenomenon. In this example, mote B selects a small t on each of its three intervals. Although other motes transmit, mote B never hears those transmissions before its own, and its transmissions are never suppressed. Figure 5.6 shows how the short-listen problem effects the transmission rate in a lossless network with $k = 1$. A perfectly synchronized single-hop network scales perfectly, with a constant number of transmissions. In a network without any synchronization between intervals, however, the number of transmissions per interval increases significantly.

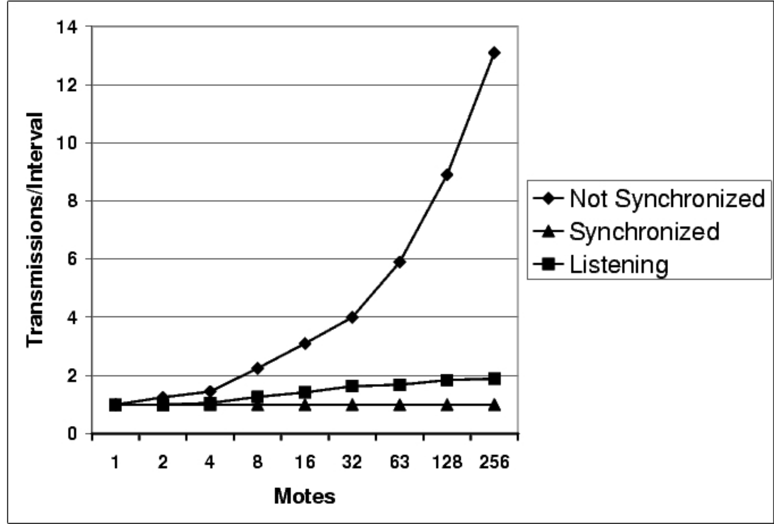


Figure 5.6. The short listen problem's effect on scalability, $k = 1$. Without synchronization, Trickle scales with $O(\sqrt{n})$. A listening period restores this to asymptotically bounded by a constant.

The short-listen problem causes the number of transmissions to scale as $O(\sqrt{n})$ with network density. [1] Unlike loss, where extra $O(log(n))$ transmissions are sent to keep the worst case mote up to date, the additional transmissions due to a lack of synchronization are completely redundant, and represent avoidable inefficiency.

Removing the short-listen effect in protocols where intervals are not synchronized requires modifying Trickle slightly.[2] Instead of picking a t in the range $[0, \tau]$, t is selected in the range $[\frac{\tau}{2}, \tau]$, defining a "listen-only" period of the first half of an interval. Figure 5.7 depicts the modified algorithm. A listening period improves scalability by enforcing a simple constraint. If sending a message guarantees a silent period of some time T that is independent of density, then the send rate is bounded above (independent of the density). When a mote transmits, it suppresses all other motes for at least the length of the listening period. With a

[1] Assume the network of n motes with an interval τ is in a steady state. If interval skew is uniformly distributed, then the expectation is that one mote will start its interval every $\frac{\tau}{n}$. For time t after a transmission, $\frac{nt}{\tau}$ will have started their intervals. From this, we can compute the expected time after a transmission that another transmission will occur. This is approximately when

$$\prod_{t=0}^{n}(1 - \frac{t}{n}) < \frac{1}{2}$$

which is when $t \approx \sqrt{n}$, that is, when $\frac{\sqrt{n}}{\tau}$ time has passed. There will therefore be $O(\sqrt{n})$ transmissions.

[2] While many protocols cannot assume synchronization, others can. For example, the Deluge protocol uses a cluster formation protocol to distribute binary TinyOS images. Part of that protocol involves roughly synchronizing nodes in a cluster to the cluster leader for the duration of the transfer, a few seconds.

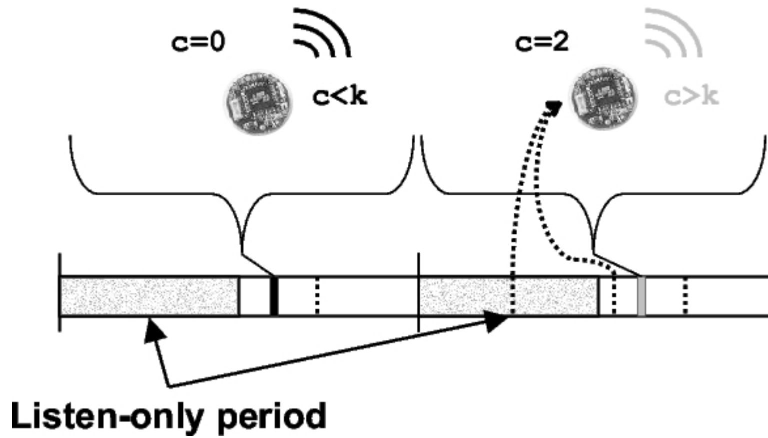
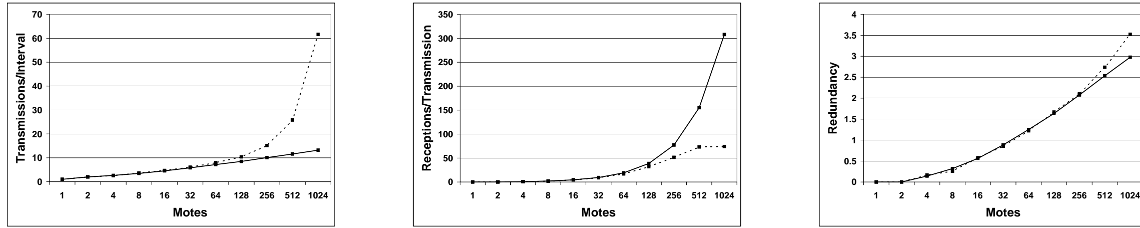


Figure 5.7. Trickle maintenance with a k of 1 and a listen-only period. Dark boxes are transmissions, gray boxes are suppressed transmissions, and dotted lines are heard transmissions.

listen period of $\frac{\tau}{2}$, it bounds the total sends in a lossless single-hop network to be $2k$, and with loss scales as $2k \cdot log(n)$, returning scalability to the $O(log(n))$ goal.

The "Listening" line in Figure 5.6 shows the number of transmissions in a single-hop network with no synchronization when Trickle uses this listening period. As the network density increases, the number of transmissions per interval asymptotically approaches two. The listening period does not harm performance when the network is synchronized: there are k transmissions, but they are all in the second half of the interval.

To work properly, Trickle needs a source of randomness; this can come from either the selection of t or from a lack of synchronization. Unless a network engages in a continuous synchronization protocol, clock drift requires any long-running trickle to use a listen-only period. The cost of a listen-only period is the distribution of the expectation of when an inconsistency will be found. Rather than the time of discovery being uniformly distributed over an interval (expectation: $\frac{\tau}{2}$, with an asymptotic minimum of 0), it is uniformly distributed over the second half of an interval (expectation: $\frac{3\tau}{4}$, with a minimum of $\frac{\tau}{2}$). For dense networks, the listen-only period can induce some detection latency. However, by using both sources of randomness, Trickle works in perfect network synchronization, no network synchronization, and any point between the two (e.g., partial or loose synchronization).



(a) Total Transmissions per Interval (b) Receptions per Transmission (c) Redundancy

Figure 5.8. Simulated trickle scalability for a multi-hop network with increasing density. Motes were uniformly distributed in a 50'x50' square area.

5.3.3 Maintenance in a Multi-hop Network

To understand Trickle's behavior in a multi-hop network, we used TOSSIM, randomly placing motes in a 50'x50' area with a uniform distribution, a τ of one second, and a k of 1. To discern the effect of packet collisions, we used both TOSSIM-bit and TOSSIM-packet (the former models collisions, and the latter does not). Drawing from the loss distributions in Figure 5.1, a 50'x50' grid is a few hops wide. Figure 5.8 shows the results of this experiment.

Figure 5.8(a) shows how the number of transmissions per interval scales as the number of motes increases. In the absence of collisions, Trickle scales as expected, at $O(log(n))$. This is also true in the more accurate TOSSIM-bit simulations for low to medium densities; however, once there is over 128 motes, the number of transmissions increases significantly.

This result is troubling – it suggests that Trickle cannot scale to very dense networks. However, this turns out to be a limitation of CSMA as network utilization increases, and not Trickle itself. Figure 5.8(b) shows the average number of receptions per transmission for the same experiments. Without packet collisions, as network density increases exponentially, so does the reception/transmission ratio. Packet collisions increase loss, and therefore the base of the logarithm in Trickle's $O(log(n))$ scalability. The increase is so great that

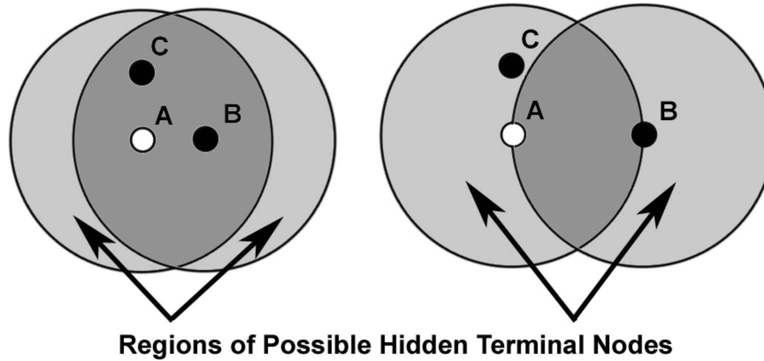


Figure 5.9. The effect of proximity on the hidden terminal problem. When C is within range of both A and B, CSMA will prevent C from interfering with transmissions between A and B. But when C is in range of A but not B, B might start transmitting without knowing that C is already transmitting, corrupting B's transmission. Note that when A and B are farther apart, the region where C might cause this "hidden terminal" problem is larger.

Trickle's aggregate transmission count begins to scale linearly. As the number of transmissions over space increases, so does the probability that two will collide.

As the network becomes very dense, it succumbs to the *hidden terminal problem*, a known issue with CSMA protocols. In the classic hidden terminal situation, there are three nodes, a, b, and c, with effective carrier sense between a and b and a and c. However, as b and c do not hear one another, a CSMA protocol will let them transmit at the same time, colliding at b, who will hear neither. In this situation, c is a hidden terminal to b and vice versa. Figure 5.9 shows an instance of this phenomenon in a simplistic disk model.

In TOSSIM-bit, the reception/transmission ratio plateaus around seventy-five: each mote thinks it has about seventy-five one-hop network neighbors. At high densities, many packets are being lost due to collisions due to the hidden terminal problem. In the perfect scaling model, the number of transmissions for m isolated and independent single-hop networks is mk. In a network, there is a *physical* density (defined by the radio range), but the hidden terminal problem causes motes to lose packets; hearing less traffic, they are aware of a smaller *observed* density. Physical density represents the number of motes who can hear a transmission in the absence of any other traffic, while observed density is a function of other, possibly conflicting, traffic in the network. Increasing physical density also make collision more likely; observed density does not necessarily increase at the same rate.

When collisions make observed density lower than physical density, the set of motes observed to be

neighbors is tied to physical proximity. The set of motes that can interfere with communication by the hidden terminal problem is larger when two motes are far away than when they are close. Figure 5.9 depicts this relationship.

Returning to Figure 5.8(b), from each mote's perspective in the 512 and 1024 mote experiments, the observed density is seventy-five neighbors. This does not change significantly as physical density increases. As a mote that can hear n neighbors, ignoring loss and other complexities, will broadcast in an interval with probability $\frac{1}{n}$, the lack of increase in observed density increases the number of transmissions (e.g., $\frac{512}{75} \rightarrow \frac{1024}{75}$).

TOSSIM simulates the mica network stack, which can handle approximately forty packets a second. As utilization reaches a reasonable fraction of this (e.g., 10 packets/second, with 128 nodes), the probability of a collision becomes significant enough to affect Trickle's behavior. As long as Trickle's network utilization is low, it scales as expected. However, increased utilization affects connectivity patterns, so that Trickle must transmit more than in an quiet network. The circumstances of Figure 5.8, very dense networks and a tiny interval, represent a corner case. As we present in Section 5.4, maintenance intervals are more likely to be on the order of tens of minutes. At these interval sizes, network utilization will never grow large as long as k is small.

To better understand Trickle in multi-hop networks, we use the metric of *redundancy*. Redundancy is the portion of messages heard in an interval that were unnecessary communication. Specifically, it is each mote's expected value of $\frac{c+s}{k} - 1$, where s is 1 if the mote transmitted and 0 if not. A redundancy of 0 means Trickle works perfectly; every mote communicates k times. For example, a mote with a k of 2, that transmitted ($s = 1$), and then received twice ($c = 2$), would have a redundancy of 0.5 ($\frac{2+1}{2} - 1$): it communicated 50% more than the optimum of k.

Redundancy can be computed for the single-hop experiments with uniform loss (Figures 5.4 and 5.6). For example, in a single-hop network with a uniform 20% loss rate and a k of 1, 3 transmissions/interval has a redundancy of 1.4= $((3 \cdot 0.8) - 1)$, as the expectation is that each mote receives 2.4 packets, and three motes transmit.

Figure 5.8(c) shows a plot of Trickle redundancy as network density increases. For a one-thousand mote– larger than any yet deployed – multi-hop network, in the presence of link asymmetry, variable packet loss, and the hidden terminal problem, the redundancy is just over 3.

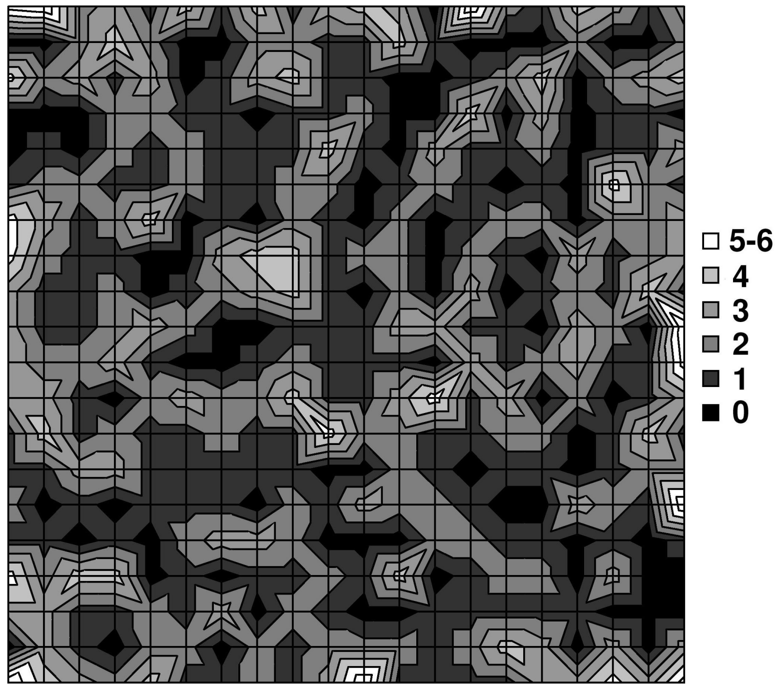
Redundancy grows with a simple logarithm of the observed density, and is due to the simple problem outlined in Section 5.3.1: packets are lost. To maintain a communication rate for the worst case mote, the average case must communicate a little bit more. Although the communication increases, the actual per-mote transmission rate shrinks. Barring MAC failures, Trickle scales as hoped – $O(log(n))$ – in multi-hop networks.

5.3.4 Load Distribution

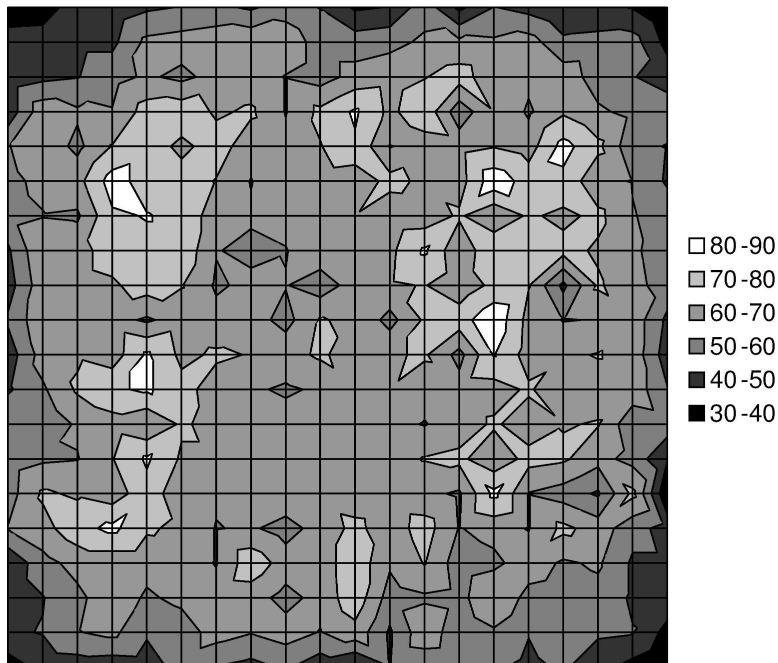
One of the goals of Trickle is to impose a low overhead. The above simulation results show that few packets are sent in a network. However, this raises the question of which motes sent those packets; 500 transmissions evenly distributed over 500 motes does not impose a high cost, but 500 messages by one mote does.

Figure 5.10(a) shows the transmission distribution for a simulated 400 mote network in a 20 mote by 20 mote grid with a 5 foot spacing (the entire grid was 95'x95'), run in TOSSIM-bit. Drawing from the empirical distributions in Figure 5.1, a five foot spacing forms a six hop network from grid corner to corner. This simulation was run with a τ of one minute, and ran for twenty minutes of virtual time. The topology shows that some motes send more than others, in a mostly random pattern. Given that the predominant range is one, two, or three packets, this non-uniformity is easily attributed to statistical variation. A few motes show markedly more transmissions, for example, six. This is the result of some motes being poor receivers. If many of their incoming links have high loss rates (drawn from the distribution in Figure 5.1), they will have a small observed density, as they receive few packets.

Figure 5.10(b) shows the reception distribution. Unlike the transmission distribution, this shows clear patterns. motes toward the edges and corners of the grid receive fewer packets than those in the center. This is due to the non-uniform network density; a mote at a corner has one quarter the neighbors as one in the



(a) Transmissions



(b) Receptions

Figure 5.10. Communication topography of a simulated 400 mote network in a 20x20 grid with 5 foot spacing (95'x95'), running for twenty minutes with a τ of one minute. The x and y axes represent space, with motes being at line intersections. Color denotes the number of transmissions or receptions at a given mote.

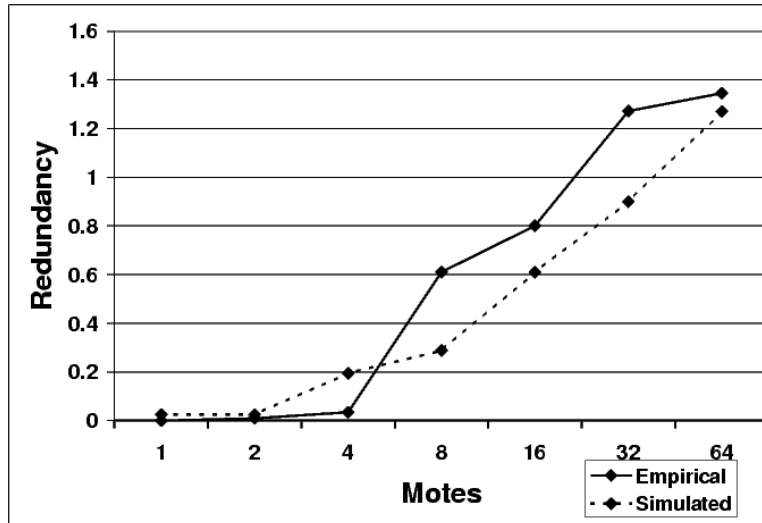


Figure 5.11. Empirical and simulated scalability over network density. The simulated data is the same as Figure 5.8.

center. Additionally, a mote in the center has many more neighbors that cannot hear one another; so that a transmission in one will not suppress a transmission in another. In contrast, almost all of the neighbors of a corner mote can hear one another. Although the transmission topology is quite noisy, the reception topography is smooth. The number of transmissions is very small compared to the number of receptions: the communication rate across the network is fairly uniform.

5.3.5 Empirical Study

To evaluate Trickle's scalability in a real network, we recreated, as best we could, the experiments shown in Figures 5.6 and 5.8. We placed motes on a small table, with their transmission signal strength set very low, making the table a small multi-hop network. With a τ of one minute, we measured Trickle redundancy over a twenty minute period for increasing numbers of motes. Figure 5.11 shows the results. They show similar scaling to the results from TOSSIM-bit. For example, the TOSSIM-bit results in Figure 5.8(c) show a 64 mote network having an redundancy of 1.1; the empirical results show 1.35. The empirical results show that maintenance scales as the simulation results indicate it should: logarithmically with regards to density.

The above results quantified the maintenance overhead. Evaluating propagation requires an implementa-

Event	Action
τ Expires	Double τ, up to τ_h. Reset c, pick a new t.
t Expires	If $c < k$, transmit.
Receive same metadata	Increment c.
Receive newer metadata	Set τ to τ_l. Reset c, pick a new t.
Receive newer code	Set τ to τ_l. Reset c, pick a new t.
Receive older metadata	Send updates.

t is picked from the range $[\frac{\tau}{2}, \tau]$

Figure 5.12. Trickle pseudocode.

tion; among other things, there must be code to propagate. In the next section, we present an implementation of Trickle in ASVMs, evaluating it in simulation and empirically.

5.4 Propagation

A large τ (gossiping interval) has a low communication overhead, but slowly propagates information. Conversely, a small τ has a higher communication overhead, but propagates more quickly. These two goals, rapid propagation and low overhead, are fundamentally at odds: the former requires communication to be frequent, while the latter requires it to be infrequent.

By dynamically scaling τ, Trickle can use its maintenance algorithm to rapidly propagate updates with a very small cost. τ has a lower bound, τ_l, and an upper bound τ_h. When τ expires, it doubles, up to τ_h. When a mote hears a summary with newer data than it has, it resets τ to be τ_l. When a mote hears a summary with older code than it has, it sends the code, to bring the other mote up to date. When a mote installs new code, it resets τ to τ_l, to make sure that it spreads quickly. This is necessary for when a mote receives code it did not request, that is, didn't reset its τ for. Figure 5.12 shows pseudocode for this complete version of Trickle.

Essentially, when there's nothing new to say, motes gossip infrequently: τ is set to τ_h. However, as soon as a mote hears something new, it gossips more frequently, so those who haven't heard it yet find out. The chatter then dies down, as τ grows from τ_l to τ_h.

By adjusting τ in this way, Trickle can get the best of both worlds: rapid propagation, and low main-

tenance overhead. The cost of a propagation event, in terms of additional sends caused by shrinking τ, is approximately $log(\frac{\tau_h}{\tau_l})$. For a τ_l of one second and a τ_h of one hour, this is a cost of eleven packets to obtain a three-thousand fold increase in propagation rate (or, from the other perspective, a three thousand fold decrease in maintenance overhead). The simple Trickle policy, "every once in a while, transmit unless you've heard a few other transmissions," can be used both to inexpensively maintain code and quickly trigger the need to propagate it.

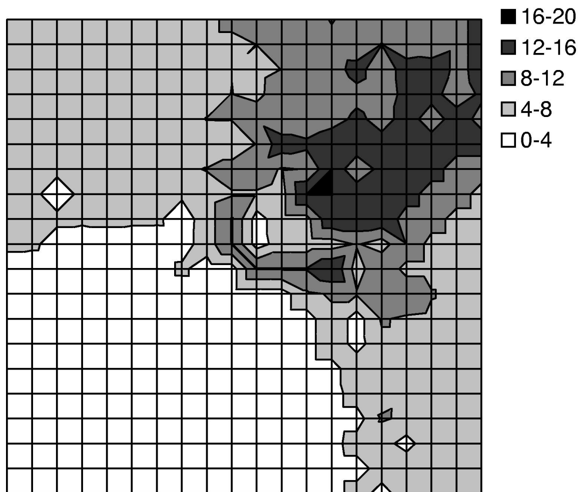
We evaluate a basic Trickle-based data propagation protocol to measure Trickle's propagation signaling latency. Using TOSSIM, we evaluate how how rapidly Trickle can propagate an update through reasonably sized (i.e., 400 mote) networks of varying density. We then evaluate the protocol's propagation rate in a small (20 mote) real-world network.

5.4.1 Propagation Protocol

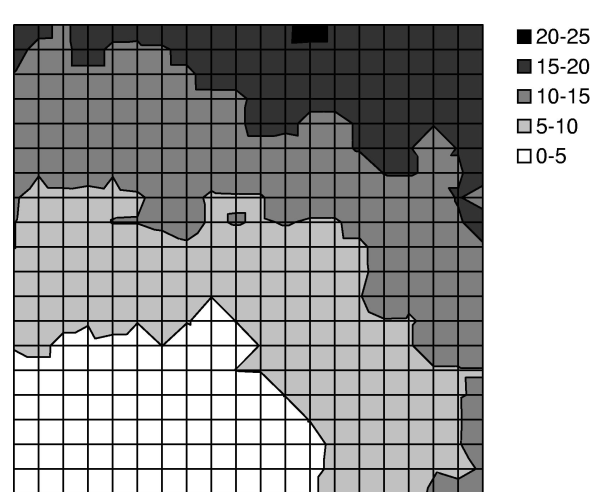
A mote maintains a small, static set of data items (e.g., code routines). Each item can have many versions, but a mote only keeps the most recent one. By replacing these items, a user can update a network's configuration or settings. Each item fits in a single TinyOS packet and has a version number. The protocol installs items with a newer version number when it receives them.

Instead of sending entire items, motes broadcast version summaries. A version summary contains the version numbers of all of the items currently installed. A mote determines that someone else needs an update by hearing that they have an older version.

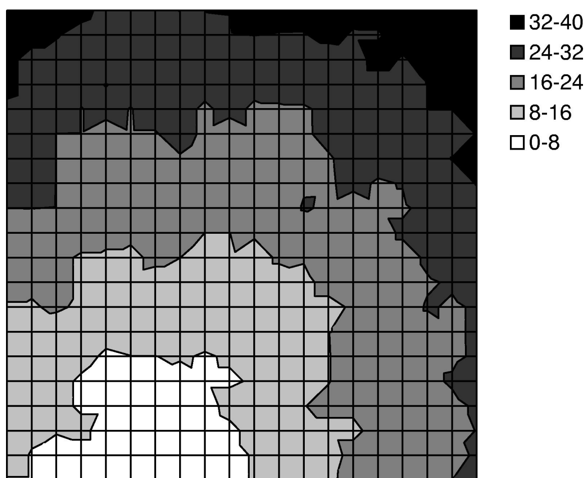
The protocol uses Trickle to periodically broadcast version summaries. In all experiments, data items fit in a single TinyOS packet (30 bytes). When the protocol detects that another node needs an update (it hears an older vector), it broadcasts the missing item three times: one second, three seconds, and seven seconds after hearing the vector. These broadcasts do not have any suppression mechanisms; they represent a very simple propagation mechanism. The protocol maintains a 10Hz timer, which it uses to increment a counter. t and τ are represented in ticks of this 10Hz clock.



(a) 5’ Spacing, 6 hops



(b) 10’ Spacing, 16 hops



(c) 15’ Spacing, 32 hops

(d) 20’ Spacing, 40 hops

Figure 5.13. Simulated time to code propagation topography in seconds. The hop count values in each legend are the expected number of transmissions necessary to get from corner to corner, considering loss.

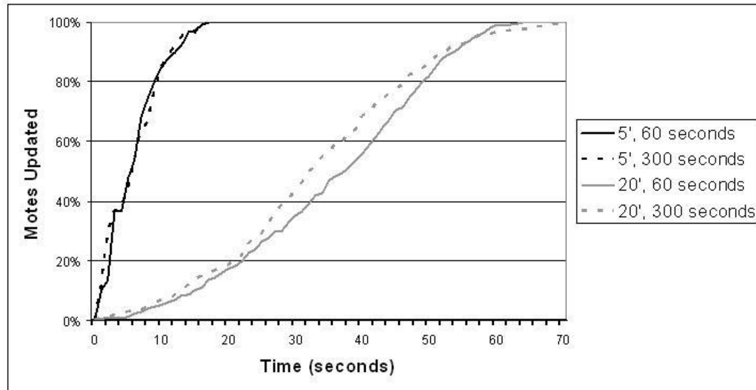


Figure 5.14. Simulated Code Propagation Rate for Different τ_hs.

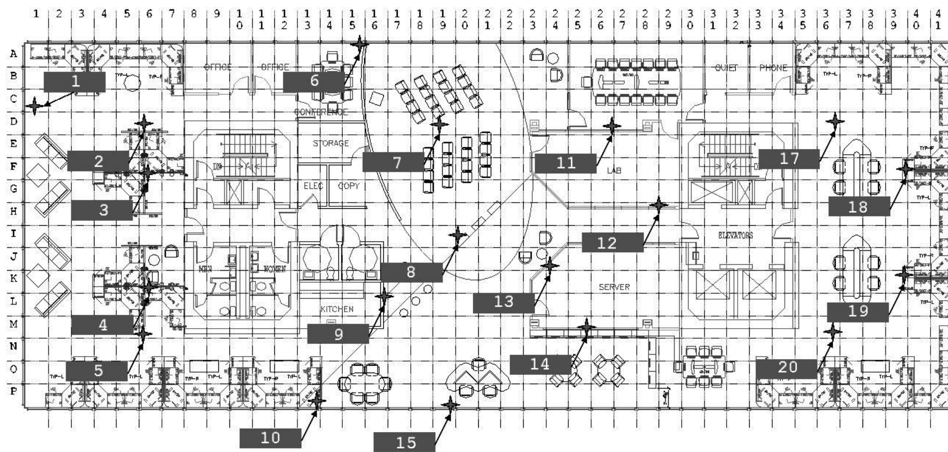
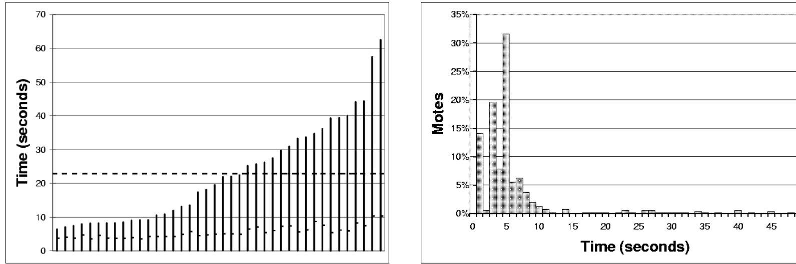


Figure 5.15. Empirical testbed layout.

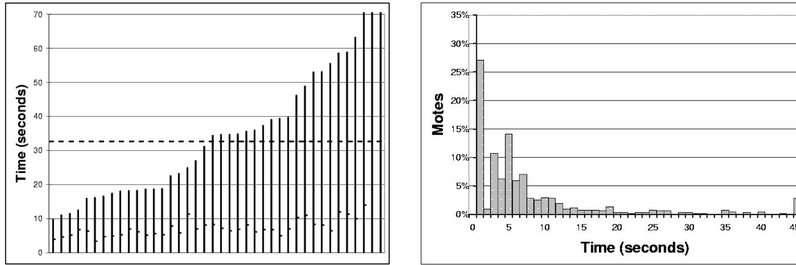
5.4.2 Simulation

We used TOSSIM-bit to observe the behavior of Trickle during a propagation event. We ran a series of simulations, each of which had 400 motes regularly placed in a 20x20 grid, and varied the spacing between motes. By varying network density, we could examine how Trickle's propagation rate scales over different loss rates and physical densities. Density ranged from a five foot spacing between motes up to twenty feet (the networks were 95'x95' to 380'x380'). We set τ_l to one second and τ_h to one minute. From corner to corner, these topologies range from six to forty hops. [3]

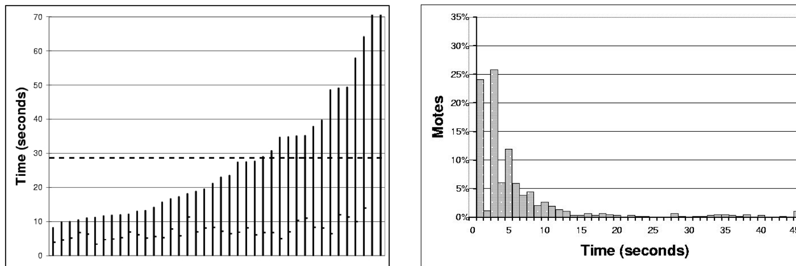
[3] These hop count values come from computing the minimum cost path across the network loss topology, where each link has a weight of $\frac{1}{1-loss}$, or the expected number of transmissions to successfully traverse that link.



(a) τ_h of 1 minute, $k = 1$



(b) τ_h of 20 minutes, $k = 1$



(c) τ_h of 20 minutes, $k = 2$

Figure 5.16. Empirical network propagation time. The graphs on the left show the time to complete reprogramming for 40 experiments, sorted with increasing time. The graphs on the right show the distribution of individual mote reprogramming times for all of the experiments.

The simulations ran for five virtual minutes. Motes booted with randomized times in the first minute, selected from a uniform distribution. After two minutes, a mote near one corner of the grid advertised a new data item. We measured the propagation time (time for the last mote to install the data item from the time it first appeared) as well as the topographical distribution of routine installation time. The results are shown in Figures 5.13 and 5.14. Time to complete propagation varied from 16 seconds in the densest network to about 70 seconds for the sparsest. Figure 5.14 shows curves for only the 5' and 20' grids; the 10' and 15' grid had similar curves.

Figure 5.13(a) shows a manifestation of the hidden terminal problem. This topography does not have the wave pattern we see in the experiments with sparser networks. Because the network was only a few hops in area, motes near the edges of the grid were able to receive and install the new item quickly, causing their subsequent transmissions to collide in the upper right corner. In contrast, the sparser networks exhibited a wave-like propagation because the sends mostly came from a single direction throughout the propagation event. This behavior is due to the lack of suppression in the propagation broadcasts.

Figure 5.14 shows how adjusting τ_h changes the propagation time for the five and twenty foot spacings. Increasing τ_h from one minute to five does not significantly affect the propagation time; indeed, in the sparse case, it propagates faster until roughly the 95th percentile. This result indicates that there may be little trade-off between the maintenance overhead of Trickle and its effectiveness in the face of a propagation event.

A very large τ_h can increase the time to discover inconsistencies to be approximately $\frac{\tau_h}{2}$. However, this is only true when two stable subnets ($\tau = \tau_h$) with different code reconnect. If new code is introduced, it immediately triggers motes to τ_l, bringing them to a quickly responsive state.

5.4.3 Empirical Study

To validate the simulation results and investigate whether TOSSIM makes simplifying assumptions that hide problems which a real protocol must deal with, we deployed a nineteen mote network in an office area, approximately 160' by 40'. We instrumented fourteen of the motes with the TCP interface described in Section 5.2, for precise timestamping. When a node installed a new data item, it sent out a UART packet; by

opening sockets to all of the motes and timestamping when this packet is received, we could measure data propagation over a large area.

Figure 5.15 shows a picture of the office space and the placement of the motes. motes 4, 11, 17, 18 and 19 were not instrumented; motes 1, 2, 3, 5, 6, 7, 8, 9, 10, 12, 13, 14, 15, and 20 were. mote 16 did not exist.

As with the above experiments, Trickle was configured with a τ_l of one second and a τ_h of one minute. The experiments began with the injection of a new data item through a TinyOS GenericBase, which is a simple bridge between a PC and a TinyOS network. The GenericBase broadcast the data item three times in quick succession. We then logged when each mote had received the data, and calculated the time between the first transmission and installation.

The left hand column of Figure 5.16 shows the results of these experiments. Each bar is a separate experiment (40 in all). The worst-case reprogramming time for the instrumentation points was just over a minute; the best case was about seven seconds. The average, shown by the dark dotted line, was just over twenty-two seconds for a τ_h of sixty seconds (Figure 5.16(a)), while it was thirty-two seconds for a τ_h of twenty minutes (Figure 5.16(b)).

The right hand column of Figure 5.16 shows a distribution of the time to reprogramming for individual motes across all the experiments. This shows that almost all motes are reprogrammed in the first ten seconds: the longer times in Figure 5.16 are from the very long tail on this distribution. This behavior is different than what we observed in TOSSIM: experiments in simulation had no such long tail.

To determine the cause of this discrepancy, we examined the logs and ran additional experiments to obtain packet transmission data. In addition to reporting when they received a data item, nodes also periodically reported how many version vectors they had received and sent, as well as how many data item broadcasts they had received and sent.

Based on these data, we determined that the long tail came from the east side of the deployment (right hand side in the figure): the only way that data could reach nodes 17, 18, 19, and 20 was a broadcast from node 14 that node 20 heard. We observed that nodes 17-20 all received the data around the same time, and

node 20 always received it first. In experiments where 20 took a long time to receive the data (and therefore 17-19 as well), node 20 send many version vectors and node 14 sent the data item many times.

Based on these observations, we conclude that the long tail is the result of the network topology. Node 20 is a poor receiver, and communication between nodes 14 and 20 represents a propagation bottleneck in the network. The combination of sparsity, a poor receiver, and a low communication rate can create these long tails. From the perspective of a network administrator, nodes 17-20 are practically disconnected from the rest of the nodes, suggesting that a few more nodes might be needed to increase the density. Additionally, while $k = 1$ works well for denser networks, where logarithmic scaling makes the actual communication rate much higher, a real communication rate of $\frac{1}{\tau}$ can induce these long tails from a just a few packet losses.

In Figure 5.16, very few motes reprogram between one and two seconds after code is introduced. This is an artifact of the granularity of the timers used, the capsule propagation timing, and the listening period. Essentially, from the first broadcast, three timers expire: $[\frac{\tau l}{2}, \tau_l]$ for motes with the new code,$[\frac{\tau l}{2}, \tau_l]$ for motes saying they have old code, then one second before the first capsule is sent. This is approximately $2 \cdot \frac{\tau_l}{2} + 1$; with a τ_l of one second, this latency is two seconds.

5.4.4 State

The Maté implementation of Trickle requires few system resources. It requires approximately seventy bytes of RAM; half of this is a message buffer for transmissions, a quarter is pointers to the Maté routines. Trickle itself requires only eleven bytes for its counters; the remaining RAM is used by coordinating state such as pending and initialization flags. The executable code is 2.5 KB; TinyOS's inlining and optimizations can reduce this by roughly 30%, to 1.8K. The algorithm requires few CPU cycles, and can operate at a very low duty cycle.

5.5 ASVM Protocol

A Maté ASVM uses the Trickle algorithm to disseminate three types of data:

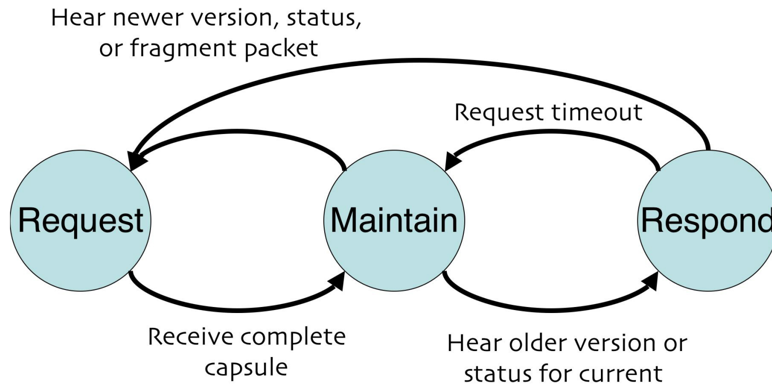


Figure 5.17. Maté capsule propagation state machine.

- **Version packets**, which contain the 32-bit version numbers of all installed capsules,
- **Capsule status packets**, which describe what fragments of a capsule that a mote needs (essentially, a bitmask), and
- **Capsule fragments**, which are pieces of a capsule.

An ASVM can be in one of three states: maintain (exchanging version packets), request (sending capsule status packets), or respond (sending fragments). Nodes start in the maintain state. Figure 5.17 shows the code propagation state diagram. The state transitions prefer requesting over responding; a node will defer forwarding capsules until it thinks it is completely up to date.

Each type of packet (version, capsule status, and capsule fragment) is a separate network trickle. For example, a capsule fragment transmission can suppress other fragment transmissions, but will not suppress version packets. This allows meta-data and data exchanges to occur concurrently. Trickling fragments means that code propagates in a slow and controlled fashion, instead of as quickly as possible. This is unlikely to significantly disrupt any existing traffic, and prevents network overload. We show in Section 5.6 that because Maté programs are small (tens or a hundred bytes), code can still propagate rapidly across large multi-hop networks (e.g., twenty seconds).

5.5.1 The Three Trickles

An ASVM handles each trickle differently. Version packets use the full Trickle algorithm: the default settings are a τ_l of one second, a τ_h of twenty minutes, and a k of 1. When a mote hears that another mote has an update, it resets the version trickle timer and transitions into the request state (if not already there). In the request state, the ASVM uses the version trickle timer software to send capsule status packets (saving RAM, since the version timer is not needed in this state). When an ASVM receives the last fragment of the needed capsule, `MVirus` returns to the maintain state and broadcasts version vectors using the version trickle timer.

When a mote hears that another mote needs an update, it transitions into the respond state and starts a separate trickle timer, the capsule timer. The capsule timer has a fixed τ of one second and a k of 3. The capsule timer runs for a fixed number of intervals before stopping (unless re-triggered by hearing another capsule status or old version packet). When in the respond state, a mote keeps a bitmask of the needed fragments from capsule status and version packets it has heard. If a mote in the respond state is not suppressed, it picks a random fragment from those it thinks are needed and broadcasts it. When the capsule timer interval expires, the mote decays the bitmask by clearing each bit with a fifty percent probability: this prevents issues like the hidden terminal problem causing two nodes to send every fragment to a single requesting node. If it turns out a fragment is still needed, a new capsule status packet will eventually indicate so.

5.6 Protocol Evaluation

Propagation energy, $P(n)$, has two parts: a maintenance cost m, which is independent of the size of programs, and a propagation cost $p \cdot n$, which depends on the size of programs. The maintenance cost exists for any system that reliably propagates code into a network. For example, the Deluge system disseminates TinyOS binaries in a network and uses a network trickle for its maintenance.

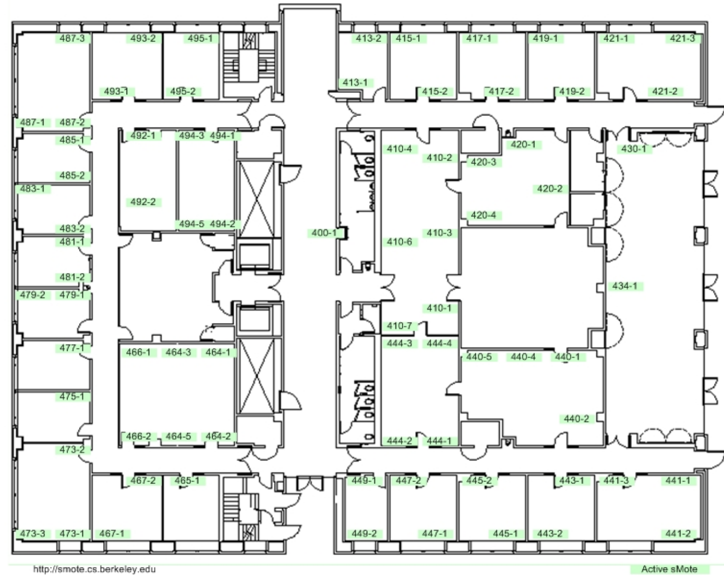


Figure 5.18. Layout of Soda Hall testbed. The code propagation experiments in Figure 5.19 injected code at mote 473-3 (the bottom left corner).

5.6.1 Parameters

By default, ASVMs use the following parameters for code propagation:

Trickle	τ_l	τ_h	k
Version vector	1s	1200s	1
Capsule status	1s	1200s	1
Capsule fragment	1s	1s	2

These parameters make the maintenance cost m equal to $\frac{log(d)}{d}$ transmissions and $log(d)$ receptions (where d is the network density) every twenty minutes. Barring tremendous densities, this cost is much lower than the idle listening cost, even under low power listening. Additionally, the cost can be adjusted to meet lifetime requirements by changing the trickle constants.

5.6.2 Methodology

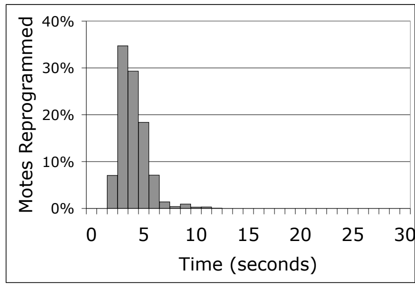
We evaluated Maté's code propagation protocol using a 70 mica2dot testbed on the fourth floor of Soda Hall at UC Berkeley. Figure 5.18 shows the physical layout of the network on the Soda floor plan. Each node in the testbed has a TCP/IP interface to its serial communication. We measured time to reprogramming and

communication costs by having nodes frequently report their state over this instrumentation back channel (10 Hz). The experiment process connects to every node and logs all of this data. We reset the network before each reprogramming event, to clear out any prior state and keep the experiments independent. The packet transmission counts are from when the code was first injected to when the last node reprogrammed: there is a slight additional cost in version vector packets as the network settles down to τ_h, but this number is small and roughly equivalent across experiments. It is not highly variable because of the network density and because the time to reprogramming is much smaller than τ_h; there is a lot of suppression and the interval skew is small compared to most τ values.

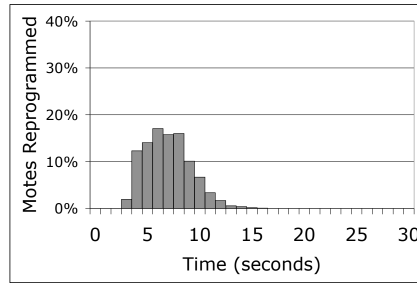
We measured the speed and cost of reprogramming for a range of capsule sizes by sending a new capsule to one node over its TCP/IP interface. The standard Maté maximum capsule size is 128 bytes (5 chunk packets), so we measured capsules of 1-5 chunks in length (10, 40, 51, 79, and 107 bytes). For each capsule size, we reprogrammed the network 100 times, for a total of 500 reprogramming events. The nodes transmission power made the network approximately four hops from corner to corner, and the code was injected on a node in the SW corner of the building (upper left in Figure 5.18).

5.6.3 Propagation Rate

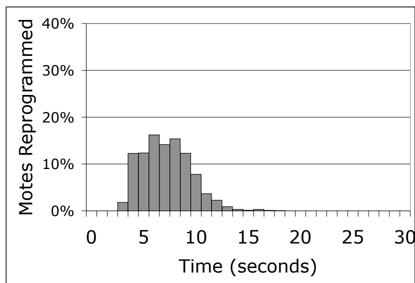
Figure 5.19 shows the distributions of reception times for each of the capsule sizes. The lack of very long tails shows that the Soda Hall network is well connected: in the 1 chunk experiments, the longest any node took to receive the new capsule (over 100 experiments) was 13 seconds. As capsule chunks are sent with a trickle, larger capsules have a significant effect on the propagation time, much more so than bandwidth considerations would suggest. This cost is due to the desire for a stable and self-regulating protocol that broadcast storms and other dangerous edge cases. The slowest time to reprogram across all of the experiments was 30 seconds.



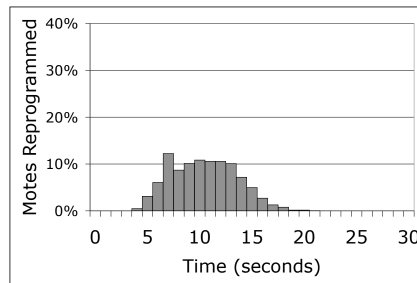
(a) 1 Chunk Capsule



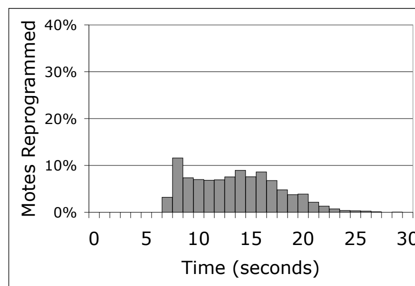
(b) 2 Chunk Capsule



(c) 3 Chunk Capsule



(d) 4 Chunk Capsule

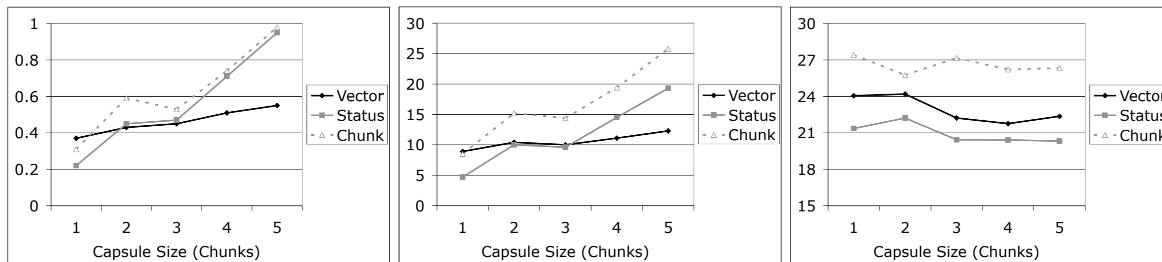


(e) 5 Chunk Capsule

Figure 5.19. Reception times for capsules of varying size. Each plot shows the distribution of reception times for the 70-node Soda Hall network over 100 experiments.

Packets	Version		Status		Chunk	
	Tx	Rx	Tx	Rx	Tx	Rx
1	0.37	8.9	0.22	4.7	0.31	8.5
2	0.43	10.4	0.45	10.0	0.59	15.2
3	0.45	10.0	0.47	9.6	0.53	14.4
4	0.51	11.1	0.71	14.5	0.74	19.4
5	0.55	12.3	0.95	19.3	0.98	25.8

Table 5.1. Per-node packet transmission and reception counts to reprogram the Soda Hall network for varying capsule sizes. These values represent the averages across 100 experiments on a 73 node mica2dot testbed.



(a) Average Packet Transmissions (b) Average Packet Receptions (c) Average Transmission Degree

Figure 5.20. Plots of data in Table 5.1. Each plot shows how the packet counts change as capsule size increases. The general trend is a linear increase. The 2 chunk capsule experiments show an aberration in this trend, with higher counts than the 3 chunk capsule. This behavior can be somewhat explained by Figure 5.20(c), which shows the degree (number of receptions per transmission) of each trickle. Some sort of network dynamic or transient interference source possibly changed the network dynamics: chunk transmissions showed a significantly lower degree than other experiments.

5.6.4 Propagation Cost

Table 5.1 and Figure 5.20 shows the number of packets transmitted and received by each of the trickles for the five capsule sizes. The packet counts are from the time when the last mote reprogrammed. After this time, there will be no more status or chunk packets, but there will be a small number of version packets above the maintenance rate as the trickle slows down to its steady state.

Depending on the size of the capsule installed, retasking the entire network requires each mote in the network to sends, on the average, 0.9 – 2.48 packets. These values show that retasking events are inexpensive, and the cost increases linearly with the size of the capsule.

5.6.5 Analysis

By using network trickles, the ASVM code propagation protocol gains all of the advantage of the Trickle algorithm: low cost, quick response, and scalability to a wide range of densities. Rather than focusing on propagating code as quickly as possible – given the program sizes involved, this could be under a second – an ASVM provides a stable and efficient protocol that reprograms large networks with large ASVM programs within a reasonable time while sending as few packets as possible.

If rate is an issue, there are several techniques that can be used to decrease reprogramming time without sacrificing scalability or stability. For example, the Firecracker protocol [65] can be used on top of an any-to-any routing scheme in order to quickly seed a network with routed packets, which can be forwarded much faster than broadcasts as there is only retransmitter. Currently, the ASVM capsule manager supports Firecracker, but by default the protocol is disabled. A user must explicitly wire an any-to-any routing protocol to the ASVM to enable it.

5.7 Security

Self-replicating code poses network security risks. If an adversary can introduce a single copy of a malicious program, he can take control of the entire network. Maté's version numbers are finite: an adversary

can cause a mote to reject all new code by using the highest possible version number. Although many sensor network applications, such as monitoring the climate in a redwood tree, do not have security requirements, others, such as art theft protection systems, do.

We have developed and implemented systems for securing epidemic code propagation under two threat models. In the first, motes are physically secure, in the second they can be compromised. Both models assume that the PC where users write scripts is part of the trusted computing base (TCB).

In the secure-mote model, the ASVM image contains a shared, private key. Using protocols such as TinySec [59], an ASVM can simply establish code integrity and/or confidentiality by using cryptographic authentication and/or encryption. Using this security model merely requires building an ASVM with a TinySec network stack, which is generally available on all TinyOS platforms.

In the physical compromise model, an adversary can examine the internal state of a compromised mote (e.g., extract keys) and arbitrarily reprogram it. In this case, motes cannot depend on one another for code. Instead, the trusted computing base signs ASVM capsules and version vectors using the BiBa algorithm [81], proving it generated the code. BiBa has the property that signatures are computationally intensive to produce, but inexpensive to verify. An earlier version of Maté supported this approach, to prove it feasible. Current ones do not, as resistance to physical mote compromise is not yet a common requirement.

Both approaches impose a small RAM (less than a hundred bytes) and CPU overhead. The CPU overhead of the first depends on the cryptographic implementation. It is negligible on more recent platforms (micaZ, Telos) whose radios include hardware encryption/decryption. The overhead of the second is also low, as BiBa signatures of Maté programs can be computed on motes in a few milliseconds. Capsules must contain BiBa signatures, which increases their length by approximately forty bytes.

5.8 Related Work

Trickle draws on two major areas of prior research. Both assume network characteristics distinct from low-power wireless sensor networks, such as cheap communication, end-to-end transport, and limited (but

existing) loss. The first area is controlled, density-aware flooding algorithms for wireless and multicast networks [36, 69, 79]. The second is epidemic and gossiping algorithms for maintaining data consistency in distributed systems [17, 28, 30].

Prior work in network broadcasts has dealt with a different problem than the one Trickle tackles: delivering a piece of data to as many nodes as possible within a certain time period. Early work showed that in wireless networks, simple broadcast retransmission could easily lead to the broadcast storm problem [79], where competing broadcasts saturate the network. This observation led to work in probabilistic broadcasts [69, 92], and adaptive dissemination [48]. Just as with earlier work in bimodal epidemic algorithms [14], all of these algorithms approach the problem of making a best-effort attempt to send a message to all nodes in a network, then eventually stop.

For example, Ni et al. propose a counter-based algorithm to prevent the broadcast storm problem by suppressing retransmissions [79]. This algorithm operates on a single interval, instead of continuously. As results in Figure 5.16 show, the loss rates in the class of wireless sensor network we study preclude a single interval from being sufficient. Additionally, their studies were on lossless, disk-based network topologies; it is unclear how they would perform in the sort of connectivity observed in the real world [63].

This is insufficient for sensor network code propagation. For example, it is unclear what happens if a mote rejoins three days after the broadcast. For configurations or code, the new mote should be brought up to date. Using prior wireless broadcast techniques, the only way to do so is periodically rebroadcast to the entire network. This imposes a significant cost on the entire network. In contrast, Trickle locally distributes data where needed.

The problem of propagating data updates through a distributed system has similar goals to Trickle, but prior work has been based on traditional wired network models. Demers et al. proposed the idea of using epidemic algorithms for managing replicated databases [30], while the PlanetP project [28] uses epidemic gossiping for a a distributed peer-to-peer index. Our techniques and mechanisms draw from these efforts. However, while traditional gossiping protocols use unicast links to a random member of a neighbor set,

or based on a routing overlay [17], Trickle uses only a local wireless broadcast, and its mechanisms are predominantly designed to address the complexities that result.

Gossiping through the exchange of metadata is reminiscent of SPIN's three-way handshaking protocol [48]; the Impala system, deployed in ZebraNet, uses a similar approach [68]. Specifically, Trickle is similar to SPIN-RL, which works in broadcast environments and provides reliability in lossy networks. Trickle differs from and builds on SPIN in three major ways. First, the SPIN protocols are designed for transmitting when they detect an update is needed; Trickle's purpose is to perform that detection. Second, the SPIN work points out that periodically re-advertising data can improve reliability, but does not suggest a policy for doing so; Trickle is such a policy. Finally, the SPIN family, although connectionless, is session oriented. When a node A hears an advertisement from node B, it then requests the data from node B. In contrast, Trickle never considers addresses. Taking the previous example, with Trickle B sends an implicit request, which a node besides A may respond to.

Trickle's suppression mechanism is inspired by the request/repair algorithm used in Scalable and Reliable Multicast (SRM) [36]. However, SRM focuses on reliable delivery of data through a multicast group in a wired IP network. Using IP multicast as a primitive, SRM has a fully connected network where latency is a concern. Trickle adapts SRM's suppression mechanisms to the domain of multi-hop wireless sensor networks.

Trickle itself does not specify how updates propagate, as the exact protocol used depends on characteristics of the data. For example, as the Drip system disseminates small items (e.g., fewer than 10 bytes), it includes the data in each advertisement [107]. In contrast, the Deluge system uses a variant of the SPIN protocol [48] to propagate large TinyOS binaries (tens of kilobytes). For large objects, the question of the representation a program takes can be made independent of how changes to programs are represented. In the sensor network space, Reijers et al. propose energy efficient code distribution by only distributing changes to currently running code [88]. The work focuses on developing an efficient technique to compute and update changes to a code image through memory manipulation.

Chapter 6

Macrosystem Evaluation: Use Cases

This chapter presents two example ASVMs, SmokeVM and SaviaVM. SmokeVM is part of a prototype fire rescue sensor network application, while SaviaVM is being deployed in Australian redwood trees to monitor sap flow rates. Each is an example of a distinct ASVM use case. SmokeVM provides a management and control interface to its application, allowing administrators to reset, test, and configure the fire detection network. SaviaVM has a data-oriented interface to its habitat monitoring application, providing mechanisms for users to adjust sample rates and request logged data. These two ASVMs illustrate how Maté's flexible boundaries enable efficient retasking, and give concrete examples of the structure and design of an ASVM's extensions. The full description file for these ASVMs are included in Appendix 9.A.

6.1 SmokeVM: Configuration and Management

Chapter 2 introduced fire rescue as a sensor network application domain. A fire rescue network performs three tasks:

- Firefighters should have information on where they are in the building, and on the structural health of their surrounding environment;

- Fire commanders should have information on structural health of the building, and on the location and status of firefighters within; and
- Building inhabitants would benefit from smart evacuation guides, which provide clear indications on the best way to exit the building in the presence of smoke and other environmental noise.

Fault tolerance is very important for a fire rescue network. Retasking core functionality, such as the calculation of exit routes, is dangerous; a program bug may only manifest in a real emergency, and could literally kill people.

SmokeVM is an ASVM for a prototype fire rescue network, implemented as a class project by ten collaborating students. As the functionality of a fire rescue is very well defined – it must meet specific requirements that will not change very often, if at all – SmokeVM provides a management-oriented interface to the network. This section is broken into three parts. The first outlines the fire rescue application's functional components and algorithms. The second presents SmokeVM, and the third describes the user interface that controls SmokeVM.

6.1.1 Fire Rescue Application

A SmokeVM node has a custom sensor board with two traffic indicators, each of which has a red, yellow, and green light. The two indicators are for two opposite directions; each has a red, a yellow, and a green light. The lights indicate the path people should take to escape a building: green means "go this way," red means "do not go this way," and yellow means "direction uncertain, proceed with caution." Yellow lights represent the case in which the network cannot definitively determine the safe path, and so defaults to the basic case of people using their own judgment. The design consideration is that sometimes requiring people to judge the safest path themselves is a much better circumstance than sending people towards danger.

A fire rescue network can be in one of two states, safe or danger. Normally, a network is in the safe state, and unobtrusively reports on the building at a slow, steady rate. However, when a fire alarm goes off, the network starts reporting at much higher rate and illuminating escape paths. When a node enters the danger state, it floods a alarm message to the rest of the network. Additionally, all periodic traffic indicates whether

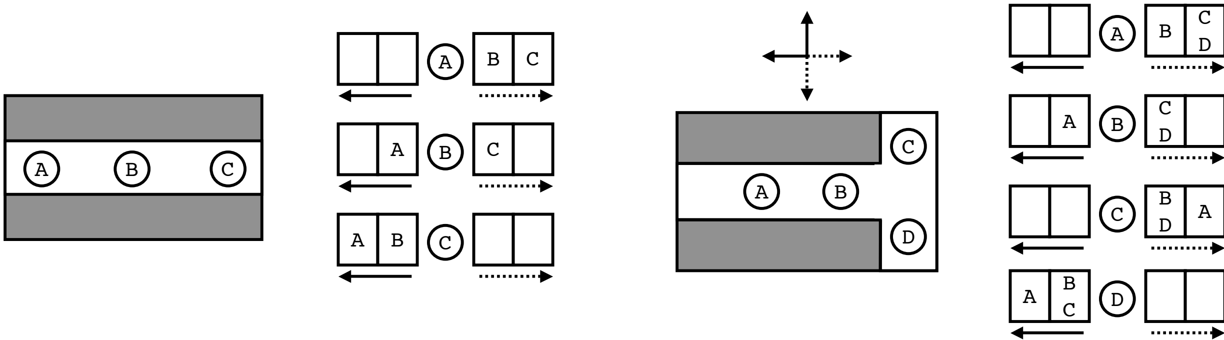
a node is in the danger state or not, and so if some nodes miss the flood they will still hear that other nodes are in the danger state and join in.

The fire rescue application meets the three domain requirements through two mechanisms. The first mechanism is a physical topology graph. Nodes use this abstraction to calculate safe escape paths; unlike a network topology, which depends on network connectivity, the physical topology depends on where the nodes are physically placed and the layout of the building. Each node has two directions, as represented by its traffic lights. The application maintains a table of physical neighbors, which are the nodes within two hops on the physical topology. Figure 6.1 shows a series of example physical topologies and how they are represented in the neighbor table. Some nodes are marked as exit nodes, representing that they are the closest point to an exit.

The second mechanism is periodic data collection, as in a periodic sampling application. The network routes data packets to a basestation, where they are logged. Nodes snoop on these data packets, to make sure that their physical neighbors are working: the application assumes that all entries in the physical neighbor table are in radio range. If a node has not heard from a physical neighbor for a few reporting intervals, it considers the neighbor absent without leave (AWOL). Each packet contains information on the state of the source node, such as

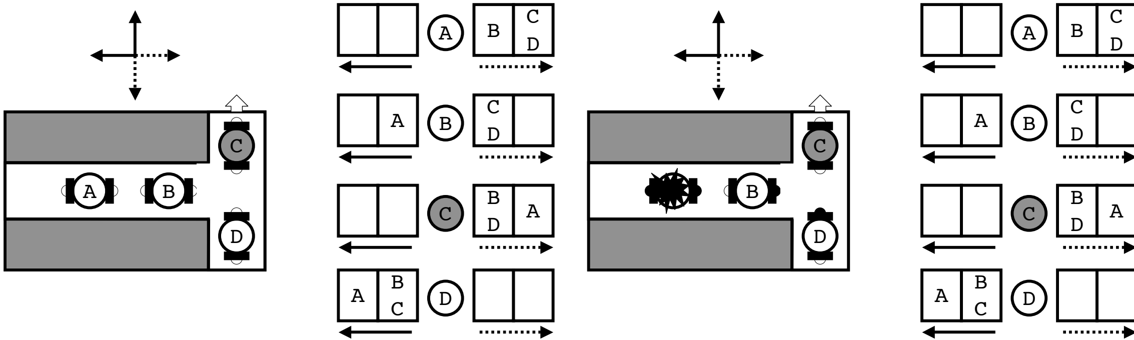
- battery voltage, for notification of when nodes need to be replaced,
- AWOL neighbors, for network troubleshooting,
- distance from an exit, for when there is a fire,
- am I in a danger state, for letting other nodes know, and
- location, for firefighter localization.

Since nodes snoop on the data traffic from their physical neighbors, they can use data packets to compute the shortest safe path to the exit. If a node cannot find any paths shorter than a reasonable distance (a constant defined by the deployment), then it does not have a path and illuminates yellow lights. If a node has detected



(a) Three nodes, A, B and C, which are adjacent in a hallway. A is two hops from C and vice versa.

(b) Four nodes at a T intersection. To B, both C and D are to the right: if either C or D is on the exit path to the exit, B will illuminate appropriately.



(c) The same topology as Figure 6.1(b), where C is an exit node and the traffic lights are shown. The network is in the safe state, so the traffic lights are off.

(d) The same topology as Figure 6.1(c), but now a fire has occurred at node A. White bulbs represent green lights, while black represent red.

Figure 6.1. SmokeVM physical topology examples. The left of each figure is the physical layout in a building. The white areas represent hallways, while the dark areas represent rooms. The right of each figure is a visualization of each node's physical neighbor table. In Figures 6.1(b), 6.1(c), and 6.1(d), the tables represent up-down rather than left-right for nodes C and D.

Name	Effect
setHealthTimer	Set the interval at which a node reports when in the safe state
bootSys	Start periodic sampling
haltSys	Halt periodic sampling
clearState	Return a node to the safe state
resetHealth	Reset period sampling
getAwolN	Fill a buffer with the AWOL neighbors
setAwolTimeout	Set how many intervals before a neighbor considered AWOL
clearNeighbors	Clear out the physical neighbor table
addNeighbor	Add a node to the physical neighbor table
setIsExit	Set whether this node is an exit (default is false)
updateNeighbor	Explicitly set a neighbor's state
onFire	Trigger a node's fire detector
stoplights	Set a particular bit pattern on the traffic lights
starttimer0	Control timer handler firing rate

Table 6.1. SmokeVM application domain functions.

a fire, then its distance is a very large number, much larger than this constant. Exit nodes (unless on fire) have a distance of zero. In the general case, a node's distance is the distance of its physical neighbor with the lowest distance plus its hopcount distance. A node with a path illuminates its lights to guide people towards the next step in the path (the green light that points towards it, the red light that points away from it). Figure 6.1 shows a simple execution of the algorithm.

These three rules mean that nodes will never choose a path that passes through a dangerous node, and will choose paths towards the an exit, illuminating it with green lights. When a node cannot figure out a safe path, it illuminates yellow.

Since nodes advertise their location, a firefighter can have a device that listens to these messages and from them estimate a physical position, using techniques similar to Microsoft's Radar project [7] or Cricket [84]. Additionally, the firefighter devices can use the existing wireless routing topology to route this position estimate them to the base station. The wireless sensor network provides the infrastructure for firefighters to know where they are in the building and report that to their commander.

6.1.2 SmokeVM

SmokeVM provides a scripting interface to the two application mechanisms with the TinyScript language. It has two handlers, reboot and a timer. Table 6.1 shows the SmokeVM functions pertaining to its

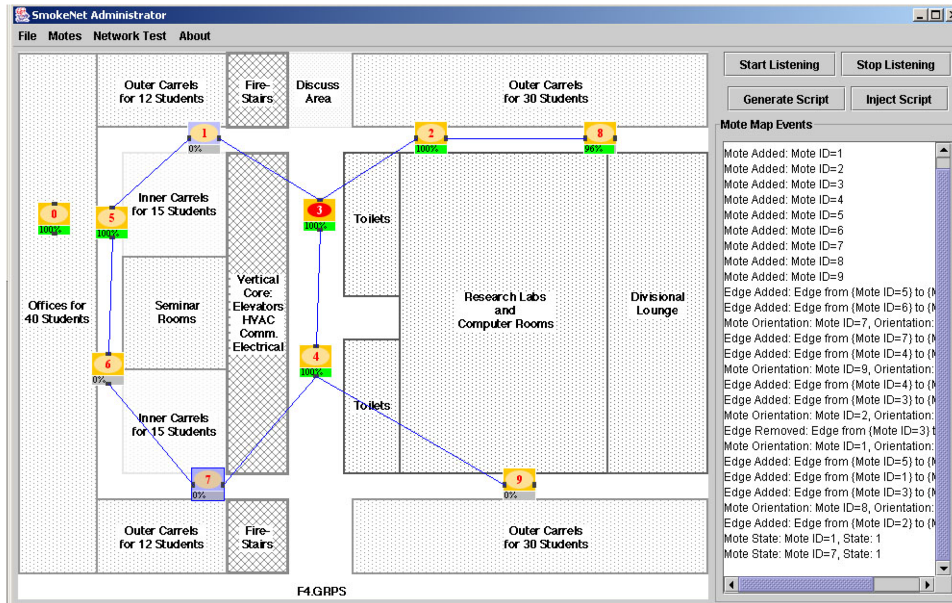


Figure 6.2. The SmokeVM user interface.

application domain. It has additional functions, for things like generating random numbers and getting the node ID.

The functions belong to three broad classes. The first class contains management functions that control how often nodes generate data packets, allowing a user to halt reporting, reset the network, and return a node to the safe state. The second class contains configuration functions for querying, modifying, and clearing the physical neighbor table. The final class contains diagnostic functions, for running tests on the network. One example test is starting the timer handler and having it display an incrementing counter on the traffic lights, to test that they are working properly.

Generally, scripts control the ASVM using the reboot handler. The timer handler is used mostly for diagnostics and tests (such as the above traffic light example). Rather than determine the algorithms used, SmokeVM provides an interface to control specific functionality. This follows the design constraint of protecting the network's safety critical operation from buggy implementations.

6.1.3 SmokeVM UI

SmokeVM provides a TinyScript execution environment, but its intended users — fire network administrators and safety testers — cannot be expected to write and compile scripts. The scripting environment allows an expert user to directly interact with the network, but SmokeVM also has a GUI interface for more general use, called “SmokeCommander” and shown in Figure 6.2

SmokeCommander listens to data packets from the collection base station and uses them to display the network status to the user. For example, if a node indicates it is on fire, the GUI colors it red, the a colored bar at the bottom of a node indicates the battery voltage, and if a neighbor is AWOL its link is shown in a different color. These visualizations allow a user to quickly determine if the network is operating correctly or if there is a problem.

When SmokeCommander receives a data packet from a node it is not already aware of, it automatically adds the node to the display in the upper right corner. SmokeCommander displays a map of the building floor and allows a user to place the sensor nodes in their physical positions. The user is responsible for specifying the physical topology of the network by connecting the two directions of a node to other nodes (the lines between nodes in Figure 6.2). The tool also allows a user to specify whether a node is an exit node; in the GUI, exit nodes are colored gray, rather than yellow.

The GUI has buttons for resetting and configuring the network. For configuration, he tool generates a series of scripts based on the specified physical topology. These scripts clear out the current neighbor tables, then set the tables of each node. The tool injects these scripts, waiting until a every node has received a script before it installs the next one. It has to install multiple scripts because often the number of nodes involved and their complexity precludes the configuration from fitting in a single capsule.

6.1.4 Analysis

SmokeVM provides a management and configuration scripting interface to a prototype fire rescue sensor network application. The network administration tool provides an even higher-level interface — a GUI — to the network, which transforms GUI commands to SmokeVM scripts. As the core application function-

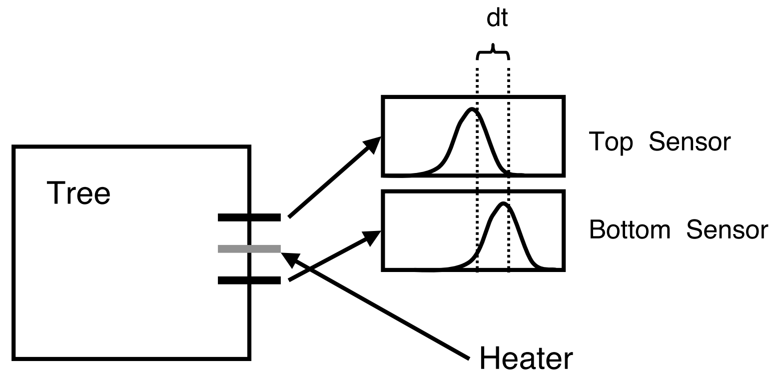


Figure 6.3. A sap flow sensor for redwood trees. The heater in the middle emits a heat pulse. Comparing when the upper and lower sensors observe the pulse provides an estimate of the sap flow rate. In this case, sap is flowing upward.

ality operates underneath the ASVM, the SmokeVM does not introduce any overhead besides some RAM utilization to store capsules. As the application does not require a great deal of data storage — the neighbor tables are approximately 300 bytes — this RAM utilization is not limiting.

6.2 SaviaVM: Redwood Data Collection

Biology researchers at the University of Western Australia are measuring sap flow rates in redwood trees. Sap flow indicates the direction and rate of a tree's fluid flow. Sap flow measurements from several points on a single tree indicates whether trees are drawing water from the soil or from fog, and whether they are releasing water into the air (transpiration). Transpiration is a critical component of climate and hydrological models, which are used for, among other things, weather forecasts.

In this application domain, small number (e.g., 10) of redwood trees are each instrumented with a few sap flow sensors. Sap flow sensors operate by emitting a heat pulse into the xylem of a tree; the sensor is two thermometers, one above the heater and one below. As shown in Figure 6.3, measuring the time difference between when the top and bottom sensors sense the pulse indicates the rate at which sap flows.

Sampling a sap flow sensor takes approximately two minutes. As these sensors generate a heat pulse, they consume a lot of energy. Rather than the standard supply of two AA batteries, motes for this application domain have a car battery, which contains approximately six hundred times the energy. Sap flow sensing so

completely dominates the energy budget of a node that standard energy conservation techniques (e.g., low power listening) are unnecessary.

Sap flow experiments require periodically sampling the sap flow sensor at a low rate, between every five minutes and every thirty minutes. For scientists to be able to determine the water sources and sinks in the tree, they need all of the sensors on a tree to sample at the same time. Additionally, to be able to compare the transpiration behavior of different trees, they need all trees to be sampled at the same time. Finally, the period of these samples needs to be regular: sampling at the same times every day allows inter-day comparisons. Being able to change the sampling period after the network is deployed would be useful; this would allow sampling for a longer period for most of the experiment duration, then sampling at a shorter period for a brief time to verify that the longer period is sufficient to capture the sap flow rate of change.

Although the network requires many forms of synchronization, the rate of change of the phenomenon being measured does not require very precise synchronization: all of the nodes in the network sampling within a few seconds of each other is acceptable.

Like most other scientific experiments, not losing data is a critical requirement. As collection routing may fail to deliver data to a base station, every data sample must be saved to non-volatile storage. Data samples that are not successfully routed to the base station can be recovered later. Being able to collect those samples without physically accessing the nodes would be very useful.

Nodes interact with the sensors over a UART-like interface. To request a sample, the mote sends a specific character sequence: when the sensor acquisition is complete, it responds with the data in ASCII text. In addition to sampling, the sensors provide a configuration interface, which allows a mote, for example, set the duration of a heat pulse. Allowing a scientists to configure and query the state of the sensors once deployed would be very useful. For example, the sap flow sensors embedded in a tree can become loose; being able to distinguish whether bad readings are due to the sensor being loose, misconfigured, or malfunctioning could save a lot of time spent traveling to and climbing trees.

In summary, a sap flow sensing network has the following requirements, in order of decreasing importance:

- Nodes in the network sample sap flow in a synchronized fashion. They must all sample within a few seconds of one another, and the period at which they sample must be regular, so that samples from different days are within a few seconds of each other (e.g., within a few seconds of 9 AM).
- Samples must be saved on non-volatile memory to ensure no data is lost.
- When a sample is taken, it must be routed to a collection point for real-time inspection and analysis.
- A user must be able to change the network's sampling period to something in the range of 5-60 minutes.
- A user must be able to remotely recover samples that are stored in non-volatile memory.
- A user should be able to configure and query the sap flow sensor remotely.

6.2.1 SaviaVM

Based on these requirements, the biologists performing this series of experiments decided to use an Telos-based ASVM. We worked with them to develop SaviaVM,[1] an ASVM for sap flow measurement. SaviaVM is a TinyScript-based ASVM with three basic classes of extensions:

1. A synchronized timer handler: when a node sets the timer to a particular period, it fires with respect to a globally synchronized time.
2. Functions to sample, query, and configure a sap flow sensor.
3. Functions that support storing samples in nonvolatile memory and retrieving them later.

The synchronized timer sits on top of FTSP, the flooding time synchronization protocol [74], which maintains a globally synchronized time across the network. When a node sets a synchronized firing period, this causes the node to trigger the handler to run when the global time modulo the period is zero. Since nodes have a globally synchronized time, this causes nodes with the same period to fire at the same time.

[1] Savia means sap in Spanish, following the etymology of Maté.

Global timestamp	
Sequence Number	
Sensor 1	Sensor 2
Voltage	Status bits

Figure 6.4. SaviaVM record format. The status bits include information such as whether the node thinks its global clock is synchronized.

Additionally, since the firing time is independent of when the period is set (it's just in term of the global clock), nodes with the same period will always fire at the same time, regardless of what their periods or firing times were previously.

The sap flow sensor functions communicate with the sensor over the Telos UART. The functions encapsulate the underlying split-phase operations. When a handler issues a command, the function implementation blocks the thread. When the reply comes back, the function packages up the result appropriately (pushes it onto the operand stack, stores it in a variable, etc) and resumes the thread. The sensor returns sample values as ASCII text of the numeric value: the sample function encodes this as a binary value which TinyScript can understand.

SaviaVM uses the non-volatile storage abstraction defined in TEP 103 [40] to allocate most of the Telos' flash chip. Each data sample is stored as a 16-byte record with the format shown in Figure 6.4. This is also the format of packet sent from the mote to the collection point. SaviaVM has functions to read stored records into a TinyScript buffer based on index in the store; a script can send the buffer to a collection point with the standard `send` function. The records have both a global timestamp and a sequence number so the scientists can reconstitute data if the network temporarily loses synchronization.

6.3 Analysis

SmokeVM and SaviaVM are ASVMs for two very different application domains. The safety critical nature of fire rescue means that SmokeVM is a management and configuration infrastructure on top of tested and static functionality. In contrast, SaviaVM is for biological experiments, so allows users to change

sampling rates, configure sensors, log, and request data. The very different uses of the two ASVMs indicate that the Maté architecture is flexible enough to support a range of application domains.

The two ASVMs provide the same programming language, TinyScript, and share many basic functions, such as `rand`, `id`, and buffer manipulation. Developers have to implement application extensions that existing library functions do not cover. This is a bootstrapping process. As more ASVMs are built, there will be a larger library of functions to draw from, requiring less effort on a developer's part. In both the SmokeVM and SaviaVM cases, developing the application logic was a significant amount of work, approximately four weeks of dedicated time in each case. Mapping them into Maté abstractions was very simple. However, this is effort that would have been necessary regardless: what Maté provides with almost no effort is a retasking system that allows a user to retask an application once it is deployed. Implementing functions required writing a small opcode component that manipulates the operand stack and calls the appropriate underlying nesC abstraction, while implementing functions required handling a nesC event. On the whole, incorporating the application abstractions in Maté took on the order of a few hours, demonstrating that using the architecture is simple.

6.4 Related Work

Sensor network deployments are still in their infancy. Correspondingly, while there has been a good deal of research on low level issues such as media access, full implementations of high-level programming systems are rare. There is one significant example, the Extensible Sensing System from UCLA.

The Extensible Sensing System (ESS) is designed to allow allow users to manage deployed sensor networks and specify triggered data acquisition [42]. The ESS approach is based on a two-tier architecture. Mote patches communicate with energy-rich *microservers*, which have high bandwidth access to a shared backbone. Microservers use a publish-subscribe mechanism based on IP multicast to allow many interested parties to monitor data. Microservers can control the data their patches generate based on client subscription requests.

In terms of data collection from mote networks, ESS depends on Directed Diffusion [56], a data-oriented

query protocol that floods *interests* through a network and routes results to the interest publishers. As its name suggests, ESS is concerned with data collection in terms of a full sensor network infrastructure, including motes, gateways, and microservers. In contrast, Maté is a system for building a programming interface to a mote network; it is more narrow than ESS in that it deals only with motes, but it is also more broad, in that it considers applications beyond data collection.

Chapter 7

Future Work and Discussion

ASVMs enable a user to safely, efficiently, and quickly retask a deployed network without adversely affecting its longevity. The Maté architecture is flexible enough to express a wide range of application domains and programming models, but makes building an ASVM simple by decomposing ASVMs into a shared template and extensions. The ASVMs achieve their efficiency by adopting a three layer programming model with a flexible network encoding layer. Customizing the encoding allows very concise programs, reducing propagation as well execution energy; the tradeoff that emerges is between expressiveness and efficiency.

Each of the preceding chapters on the Maté architecture noted and discussed prior work related to their content. These focused comparisons established that Maté is a novel approach which satisfies requirements that prior work does not. A broader comparison is also warranted: mote networks are a new domain for systems and networking research. Examining ASVMs in the larger scope of operating systems and networking provides insight on mote network research as a whole.

This chapter discusses ASVMs and Maté in regards to prior and existing work in related areas, such as operating systems, network dissemination, distributed systems and parallel programming. Comparing them with this corpus demonstrates how mote networks are distinct from these other domains and provides insight on the basic design considerations for mote network systems and protocols.

7.1 Operating Systems

A traditional desktop/server operating system has a monolithic kernel that provides a process abstraction to user programs. The kernel virtualizes underlying hardware resources, presenting user programs with the illusion of having complete use of the system's resources. Hardware mechanisms, such as protection levels and virtual memory page tables, provide the basis for these abstractions. Sensor mote MCUs do not have such resources, so virtualization must instead be based on software mechanisms: rather than the user-land abstraction provided by a kernel, mote networks have a virtual-land abstraction provided by an ASVM.

While the traditional kernel is monolithic, OS research has proposed a wide range of other approaches to deal with its perceived limitations. For example, a monolithic kernel exposes a hard and static abstraction boundary to underlying hardware resources. This hard boundary limits system efficiency, as it must be general enough to support a wide range of applications even if the system only runs one (e.g, a web server). This observation has led to many proposals for how to easily compose or customize an OS interface and implementation. As ASVMs supplant a traditional kernel as the protection mechanism, analyzing the distinctions between the approaches illustrates the differences between the domains and sheds light on sensor network systems in general.

7.1.1 Virtualization

Virtualization provides two services to user programs: protection and resource management. Virtualization provides protection by separating a program from an actual hardware resource. For example, rather than deal with a real processor that issues interrupts corresponding to hardware devices, a UNIX program runs on top of a virtualized processor that issues signals. Accessing real hardware resources is dangerous, as they generally do not have protection mechanisms.

Separating a program from a real resource also allows an OS to manage and schedule of the resource between several programs. For example, while a UNIX program might be able to access a raw hard drive through the `dev` filesystem, reads and writes to the drive are interleaved with those of other programs. This can, for example, improve overall system efficiency by clustering reads and writes by disk proximity [75].

Providing the illusion of exclusive access to a resource can greatly simplify programs. Rather than deal with the case in which another program currently has the resource and explicitly retry, the kernel makes this behavior implicit. On one hand, the user program is simpler; on the other, sources of latency are hidden, and this makes high performance systems more complex. Single use systems such as databases or Internet services benefit from being able to directly manage hardware resources: virtualization can interfere in their ability to do so [110].

7.1.2 User land

In a traditional kernel, memory and CPU are the two fundamental OS virtualizations that form the abstraction of a process. The kernel presents other resources through system calls. Virtualizing the processor allows the kernel to multiplex several concurrent programs on a single physical CPU, presenting the illusion that each of them runs interrupted. Similarly, virtualizing memory allows each program to view the address space as if it had access to the complete memory of the machine, even beyond what is available. Presenting user-land depends on hardware support for translating virtual to physical addresses.

The perception of complete access to the memory space of the machine, or user-land, also protects user programs from each other. Barring mechanisms such as shared memory regions, the full address space is private to a process, and so two processes share no addresses.[1] User-land also protects the system as a whole: a user program cannot access the shared state that the kernel represents. Encapsulating processes encapsulates their failures. A buggy program cannot cause another to fail, and more importantly, cannot cause the kernel — the system as a whole — to fail.[2]

User-land allows a system to run several independent binary programs concurrently. Because hardware automatically translates virtual addresses to physical ones, user codes run as native binary instructions at full machine speed. The only overhead is when the hardware requires an update to the virtual to physical

[1] Barring some edge cases, such as the read-only zero page. It is debatable whether each process has its own zero page, or whether there is a single shared one.

[2] Of course, a user program can trigger kernel bugs that cause the kernel itself to fail. Also, the kernel API might have exploits that allow a program to monopolize or overutilize the system: the archetypal example is forking an infinite number of new processes.

mapping (i.e., TLB) so it can translate addresses. This functionality is the basis of a multi-tasking, general purpose operating system.

7.1.3 Customizability

As the traditional hard boundary is not well suited to high performance, single use systems, one way to better support them is to have an OS with a flexible and customizable boundary. In this model, the OS provides very low-level abstractions (often not fully virtualized) and an application extends the OS by building the higher-level abstractions that it needs. One of the major challenges to such an approach is safety. Extending the OS such that it violates expected protection policies is undesirable, as isolating failures is highly beneficial.

Microkernels use message-passing and user-level servers to obtain extensibility. A microkernel OS, such as Mach, tries to keep the amount of code that runs in supervisor mode to a minimum [86]. The microkernel provides virtualized hardware abstractions, such as network interfaces and disks. Higher level abstractions, such as files, are implemented in user-level processes that build on top of the microkernel abstractions. When a user application wants to access an application-level abstraction (such as a file), it passes the request to the microkernel, which dispatches it to the appropriate server. By placing the extensions in separate processes, microkernels use memory virtualization as a protection mechanism.

While microkernels are a clean solution, they suffer from poor performance, predominantly due to the number of protection boundary crossings: a single system call has to make four, rather than two, crossings, and involves changing address spaces [53]. Academic positions and visions aside [11], this problem remains today [76]. While this overhead is tiny for system-call light codes, it has still led microkernels to be largely seen as ineffective or best left to academic investigation [47]. Windows NT, for example, began as a micro-kernel and quickly incorporated more and more functionality into the kernel, until it more closely resembles an extensible monolithic kernel.

The failure of microkernels led to their successors, SPIN and Exokernel. SPIN provides a OS which programs can extend by directly loading code into the kernel: language type safety (Modula 3) provides the

needed protection [10]. Exokernel, in contrast, eschews the notion of loading shared extensions. It provides direct hardware abstractions, which applications build directly on top of by including a ”library OS” that provides the needed application-level abstractions [58]. On one hand, these approaches show markedly improved performance over a monolithic kernel. On the other, they require a large degree of coding and engineering effort needed to obtain that performance. Another solution, which is much simpler and cheaper, is to just use more hardware.

7.1.4 How Mote Systems Differ

Mote networks are distinct from the traditional OS domain in three key ways. First almost all networks are application specific: exceptions include deployments such as experimental testbeds. Second, unlike traditional systems, where more hardware can solve performance problems, mote networks are limited by energy: adding more hardware has a cost. Finally, the lack of a long history of systems development makes their development less hindered by long-held sensibilities and corporate pressure (e.g., the weight behind monolithic kernels).

As application-specific operation is the norm rather than the exception, general purpose systems lose much of their appeal. In traditional computing domains, general purpose systems deal well with the common case and require inventive engineering for specialized uses. In contrast, general purpose mote network systems are inefficient and constraining in the common case, hence the extensible component models of platforms such as EmStar [43] and TinyOS [67].

In traditional computing domains, additional hardware resources can overcome the inefficiencies introduced by a general abstraction boundary. This costs money, but much less than developing a new operating system. However, in mote networks, this approach introduces an additional cost: energy. Faster processors, more RAM, and higher bandwidth radios cost power. A generic abstraction boundary leads a network to either require larger batteries, increasing form factor and obtrusiveness, or to sacrifice lifetime.

In traditional systems, UNIX has been — for good or for ill — the measure against which other operating systems are judged. On one hand, the reduced performance microkernels exhibit precluded them from

gaining widespread use; on the other, exokernels demonstrate large performance gains, but the increased complexity prevents adoption of their techniques. Commercial pressures for standardization (e.g., POSIX), as well as conventions of the importance of various tradeoffs lead monolithic systems to be the preferred approach. Sensor networks, however, have neither of these considerations. Historically, embedded computing has been a very fractured market: limited resources and the comparative simplicity of operation makes this feasible. As mote networks are a relatively new field, they are free of much of the baggage, backwards compatibility, and commercial entrenchment observed in traditional systems.

The confluence of these three characteristics suggests the directions future mote systems work will take. Rather than historical OS research, which has searched for the "right" abstraction boundary to a resource,[3] mote systems research is about designing systems that have flexible boundaries while providing enough structure that using them is not prohibitively difficult. For example, Rather than specify a particular virtual boundary, ASVMs allow a user to customize the boundary to a deployment's needs.

7.2 Network Protocols

Seven principles guide the design of the Internet architecture [22], the foremost of which is to provide an "effective technique for multiplexed utilization of existing interconnected networks." One ramification of this design principle is a focus on end-to-end communication. The observation is that as connectivity is the most important result the Internet can provide, and pushing state and complexity to the edge of the network is desirable [19].

Pushing state and complexity to the edge follows the more general end-to-end systems argument [91]. While intermediate processing can improve performance, ultimately the communication end points must be responsible for completely and correctly implementing the desired functionality. In the world of the Internet, where communication patterns are based on edge hosts, the end-to-end principle has significant weight and power.

[3] Take, for example, threads. The set of threading models — one to one, many to one, many to many — as well as the kernel interface to their used (fork() vs. clone() vs. LWPs in older versions of Solaris vs. scheduler activations in HP UNIX) are all investigations of finding the right interface to a well-defined OS abstraction.

Mote networks, however, have different communication patterns, due to their different resource tradeoffs and application domains. In mote networks, every node can be both an endpoint and a router; rather than the two-tier architecture the Internet follows (end points and routers), mote networks have a model much closer to peer-to-peer overlays such as Chord [100]. Unlike the Internet, which is based on a single multihop (network) protocol, IP, mote networks use many multihop protocols, such as collection, dissemination, flooding, aggregation. This delegation of responsibilities leads multihop protocols in mote networks to have a very different set of design requirements than protocols for IP devices.

7.2.1 Metrics

Mote protocols generally have two basic metrics: energy and state. A protocol that consumes less energy (e.g., through less communication) is better. As RAM is such a valuable resource, reducing the amount of state a protocol requires is also desirable. These two metrics often present a tradeoff in design. For example, maintaining more state (e.g., a data cache) is a common technique for reducing packet transmissions.

These two metrics are different from those observed in traditional IP networking, which is generally concerned with throughput and latency. For example, adding more state to Internet protocol packet headers can be justified if the state overall increases data throughput; in contrast, for mote networks the justification would be that the increased state leads to an overall reduction in energy consumption.

7.2.2 Communication Patterns and Administration

The Internet has a single network (multihop) protocol, IP, which provides best effort end-to-end packet transmission. While IP multicast does exist, it is neither widely deployed nor used today. One issue that deploying multicast faces is state: maintaining multicast trees requires storing state within the Internet routing infrastructure. While doing so is possible, the amount of state needed, combined with a general skepticism of pushing complex logic into the network core, is one of the stumbling blocks to multicast's full deployment [31].

End-to-end communication between pairs of hosts supports the notion of the Internet being a shared

resource for many parties. Many services and users can independently communicate with each other with the shared routing infrastructure.

Rather than many independent uses, mote networks commonly have a small number of collaborative uses, where all of the nodes in a network work together towards a common goal. Correspondingly, they exhibit a range of network protocols, rather than one. For example, dissemination is a class of network protocols that reliably delivers a piece of data to every node in a network, collection is a class of network protocols that delivers data from nodes to a collection root, and aggregation is a class of network protocols that computes high-level summaries of whole network data.

The Internet is a collection of Autonomous Systems (ASes) controlled by separate parties: each AS can generally be considered a separate administrative domain. Routing is predominantly a question of which ASes will peer with one another. Imposing in-network security policies is difficult, as they can require collaboration and agreement between different ASes. In contrast, a mote network generally represents a single administrative domain under the control of a single party. This greatly simplifies issues such as resource management and security.

7.2.3 Network State and Administration

The end-to-end Internet model seeks to minimize in-network data traffic state. There is, of course, a lot of state pertaining to routing and other infrastructure services, but this state pertains to the Internet control plane, rather than its data plane. Barring a few complexities such as NATs, the meaning of end-to-end in IP is clear.

In mote networks, the notion of end-to-end is less clear. Take, for example, aggregation such as the TinySQL statement “SELECT avg(light).” This query could be implemented using Synopsis Diffusion [78], which merges order and duplicate insensitive aggregates on a per-hop basis towards the query collection point. For each piece of data transmitted (a node’s synopsis), the end-to-end path is a single hop.

This dependency on per-hop transmissions and transformation naturally leads to in-network state. Maintaining state within the network can greatly improve its energy efficiency. Rather than depend on perfect

results — e.g., reliably collecting data from every node and aggregating at the root — sensor networks must approximate results, due to the kinds of loss and network transience low power radios exhibit. As sensor networks measure the real world, rather than focus on abstract data, this approximation is acceptable. For example, while the least significant bit being prone to errors would be problematic in a text document, in a sensor network the loss of precision could be irrelevant with respect to the accuracy of the sensors sampled.

7.2.4 Design Considerations for Mote Protocols

Both existing techniques that Trickle draws from – broadcasts and epidemics – have assumptions that make them inappropriate to problem of code propagation and maintenance in sensor networks. Trickle controls the radio channel in a manner similar to scalable broadcasts, but continually maintain consistency in a manner similar to epidemic algorithms. Taking advantage of the broadcast nature of the medium, a sensor network can use SRM-like duplicate suppression to conserve precious transmission energy and scale to dense networks.

Ultimately, however, Trickle is an attractive algorithm because of its simplicity. Its simplicity allows it to operate well in a wide range of network topologies. Barring edge conditions that are fantastically unlikely to occur in the real world, and if they did, would completely break almost any form of communication (e.g., a directed bipartite graph), it will signal propagation efficiently. Moreover, the simplicity of the algorithm makes it very easy to reason about and understand its behavior, even in deeply embedded systems. The scale, distribution, and embedment of mote networks means that they must be, for the most part, self-organizing in their protocols and systems [35]. Trickle is such a protocol.

Rather than work very hard to compute minimum covers of the topology or set up complex packet exchanges, Trickle takes advantage of randomization and idempotent operation. Although it is possible that a given time interval might have a bad selection of transmitters, such an event is unlikely, and more importantly, even if it happens once, it remains unlikely. Randomization avoids obscure corner cases from leading to deterministically poor performance.

The Trickle algorithm has a single external input, receiving a packet. If it receives the same packet twice,

its behavior is the same. As there are no multi-part packet exchanges, an implementation does not need to keep a lot of state, and cannot enter disconnection edge cases which require timeouts. The loss rates observed with low power radios will inherently lead to transmission redundancy from false negative suppression or acknowledgment. However, unlike Internet systems, where a typical approach to allow packet idempotency is to push further state into the two end points (e.g., NFS [80]), this is not as trivially done in the limited memory storage of a mote.

7.3 Programming Models

The end-to-end model of the Internet has constrained network programming to being a concern of edge hosts. While infrastructure routers run software, have operating systems, etc., they are generally proprietary artifacts that are outside the reach of an end user. Unlike this class of systems, where nodes are either data sources or sinks, in mote networks a node can be a data source/sink and a router, controlling how data flows through the network.

7.3.1 Parallel Processing

Given the importance of in-network processing, this distinction has significant ramifications to mote network programming. Rather than Internet end hosts, a closer first analogy for mote programming is processing in parallel architectures [29]. In this domain, processors pass messages to communicate the data needed for computation: the active messages of TinyOS are inspired by the original active message work for parallel architectures [109]. Similarities exist beyond the notion of a collection of independent computational elements that communicate. For example, timing issues in multihop communication [61] on motes have direct correlations in timing processor messaging [16].

This resemblance can also be seen in regions programming [111], which was described in Section 3. The abstraction of shared data spaces into which nodes can put values and apply reductions exists in parallel programming, and is the most commonly used programming abstraction: MPI (Message Passing Interface) [37].

The analogy to parallel processing, while helpful, can also be misleading. The principal distinction between parallel architectures and mote networks is the nature of their communication channels: the former relies on reliable (lossless) messaging to a dedicated set of processors, while the latter has a wildly variable and lossy wireless channel. Algorithms which are feasible in parallel architectures, such as a simple barrier, can be complex in mote networks, as nodes can enter and leave the network due to interference. For example, the regions program first proposed in [111] and presented in Section 3 has a race condition, which we discovered while trying to run the regions code distribution. Look at these five lines:

```
sum = region.reduce(OP_SUM, reading_key);
sum_x = region.reduce(OP_SUM, reg_x_key);
sum_y = region.reduce(OP_SUM, reg_y_key);
centroid.x = sum_x / sum;
centroid.y = sum_y / sum;
```

Each of the three reductions is a separate operation, but the final value computed (the centroid) depends on all of them. When a node applies a reduction to a region, it must some times deal with stale data, either because a node has not replied with a current value (in a pull model) or because it has not heard the most recent update (in a push model). Locally, however, a node computes its `sum_x` and `sum_y` based on the most recent value. This can lead to wildly incorrect values. Consider this example:

1. Mote M at (5, 7) notices the vehicle with a small sensor reading (10). It updates its values in the regions: reading: 10, x: 50, y: 70.
2. The mote that will do a reduction receives those values successfully.
3. Mote M notices the vehicle with a larger sensor reading (50). It updates its values in the regions: reading: 50, x: 250, y: 350.
4. The mote that will do a reduction fails to receive the reading update, and performs a reduction: mote M's inconsistent values will skew the computation, as with a perceived reading of 10, it will be included in the computation as if its position were (25, 35) rather than (5, 7).

In a lossless communication medium such as what MPI is designed for, this problem cannot occur; in

mote networks, however, it is common. Anecdotally, we observed this sort of data race to cause ten-fold swings in the position estimate regions would produce.

7.3.2 Declarative Languages

Declarative languages such as TinySQL present another approach to programming mote networks. Unlike parallel processing proposals, which are basically imperative programs with a communication library, TinySQL programs have no notion of communication, nodes, or neighbors. On one hand, more declarative languages can be simpler, and can leave optimization decisions to a compiler or runtime that have knowledge a programmer may not: all programmers can benefit from hard optimization work that is only done once. On the other hand, by pushing optimization decisions into the compiler and runtime, a more declarative language constrains a knowledgeable user into the optimizations its toolchain knows about.

Pushing optimization to be an obligation of the compiler and runtime can have additional drawbacks. Developing mote software is notoriously difficult, due to embedment, scale, and event-driven execution. Pushing logic into the language runtime increases its complexity, similarly increasing the chances of bugs and failures. For example, in the one real deployment of TinyDB (redwood trees in California's Sonoma County), 65% of the motes never reported a data value to the base station. A post-facto analysis concluded that a combination of interactions between time synchronization and TinyDB's routing as well as an inability to form a routing tree caused the failures citesnms. Given that the network ran a single query that was known at deployment time, a much simpler node runtime would have sufficed.

7.3.3 Active Networks

Section 3.6 and Section 4.7.5 pointed out the similarities and differences between active networking and mote network retasking. While the principles — push processing as close to the data source — are the same, the use cases and requirements of the domains lead to completely different solutions.

One goal of active networking was to be able to easily deploy new protocols in the Internet [105]. ANTS [112], for example, proposed an API for generating, modifying, and forwarding packets. This API

is essentially an ASVM for IP packet processing. In the Internet domain, the scope of active network's use is quite narrow, so a small number of ASVMs would suffice; in mote networks, however, the diversity of application domains requires a way to be able to build a wide range of ASVMs.

7.3.4 Analysis

The Maté architecture quite deliberately eschews taking a stance for a particular programming model: it is unlikely that there is a silver bullet programming model for all application domains. By separating the user language and network encoding, ASVMs enable flexible mappings between the two.

For example, the implementation of TinySQL presented in Section 4.7.3, rather than push query planning into the network itself, retasks the network with an imperative query plan (motlle code). For simple queries — which are the common case in deployments today — this does not sacrifice optimization opportunities that a more declarative program might allow (there is only so much optimization that can be applied to a simple SELECT statement). However, if a domain needs an ASVM that supports complex in-network optimizations, an appropriately more declarative primitive and function set could be designed. Such a design would be an interesting research problem: one straightforward solution would be to use operations that effectively interpret TinyDB queries.

This example demonstrates how the ability to separate the user program and network encoding is beneficial: in either case, users can program the network with TinySQL queries, but the underlying runtime can be optimized for the expected classes of queries expected. In a similar vein, a programming model such as regions could be readily compiled to an ASVM which, rather than directly support regions operations, provides other communication primitives, such as neighborhoods [113].

Evaluations of some of the existing mote programming proposals — TinySQL and Regions — show that there is a discontinuity between research vision and reality. Regions takes the model of shared tuple spaces and MPI-like reductions, but having unreliable rather than reliable messaging introduces data inconsistencies that can be very problematic. Similarly, while TinySQL's declarative statements allow a mote runtime to make many optimizations, for basic queries, which are the common use case today, all of the

machinery needed for these optimizations is unnecessary and imposes a large energy burden. Additionally, the complexity that a full declarative interpreter requires increases the chances of software failure, as made evident by the collection success rates of the real world deployment of TinyDB in redwood trees.

7.4 Discussion

The prior analyses indicate three important trends in wireless sensor network research: flexible boundaries, simple randomized protocols, and data models that do not depend on strict consistency. ASVMs built with the Maté architecture provide the first two. The last one is an important requirement for future programming models that are built on top of ASVMs.

The fault tolerant and flexible in-situ retasking ASVMs provide greatly outweighs their cost of using a moderate amount of RAM and code space. At no measurable detriment to the lifetime of a network, a user can quickly, safely, and easily retask a deployed network in a high level language. There has been a lot of discussion on *what* that language should be, but very little on how a given language can be realized. Experience has shown that elegant high-level models can be simple to the user, but complex and difficult to make work well. TinySQL provides many opportunities for optimization, but a system that tried to take advantage of many of those opportunities, TinyDB, failed to work well in a real deployment.

The Maté architecture applies systems and software engineering techniques, such as composability, encapsulation, and well defined interfaces, to this problem. It provides a decomposition and structure to a hard systems problem, building a runtime for high level languages. CPU efficiency is the common reason developer build custom solutions, rather than work within a generic framework; in mote networks, CPU efficiency, while important, is not a critical optimization goal. As the energy results show, this means that the costs common to generalized solutions are irrelevant in this domain.

This means that rather than just being an academic curiosity, ASVMs are a compelling solution to an open and important problem in mote networks today. In-situ retasking will let these networks transition from static, data collection systems to dynamic systems that process data within the network, reducing traffic and thereby lengthening lifetime. The barrier to entry for deploying new applications will be much lower, as

users will be able to program networks in safe, high-level languages, rather than worry about low-level hardware issues. Application specific virtual machines make mote networks accessible to those who use them, creating much greater possibilities of deployment, innovation, and application.

7.5 Future Work

The Maté architecture allows users to easily build ASVMs, and ASVMs allow users to quickly, safely, and efficiently retask deployed mote networks. However, the architecture makes several simplifying assumptions, which future retasking needs may find problematic. Depending on how application requirements evolve, these could suggest areas for future work.

7.5.1 Execution and Data Model

Currently, the ASVM template does not provide a call stack. If a language, such as motlle, wishes to provide subroutines, then it must implement its own call stack data structure. Including a it as part of the template requires designing and implementing an abstraction that can work well in many programming languages. The absence of a generic call stack is predominantly due to complexities posed by a run-time allocation mechanism and the failures it might cause.

Currently, the ASVM concurrency model has only basic mutual exclusion locks. Depending on the degree of concurrency needed, introducing more complex synchronization primitives – such as read/write locks – could improve system performance during bursts of network activity.

Section 4.7.3 showed that, due to the low duty cycle typical of mote applications, the interpretation energy overhead ASVMs impose is negligible. However, there are circumstances where interpretation efficiency is important, such as dealing with time critical tasks. Improving the efficiency of the Maté scheduler would allow handlers to be pushed down to events that are more timing sensitive. The common technique for efficient interpretation, threaded code [8], would require some modifications to remain flexible enough

for ASVM construction, but the interpretation overhead could be reduced by at least a factor of five with some restructuring.

7.5.2 Error Reporting

Currently, when an ASVM encounters a run-time error, it halts all execution, blinks its LEDs and broadcasts the conditions of the error. This behavior is fine for attended networks, but for networks to be mostly unattended and administered remotely, better error reporting mechanisms are needed. The ability to log and scan error reports (which need not be fatal) would be useful for long-term network maintenance. Fully instrumenting the ASVM with a system such as SNMS [107] could be a first step in this direction; recent work on systems such as DTrace [18] suggest the requirements and flexibility such a system could meet and provide.

7.5.3 Heterogeneity

Maté assumes that deployments are homogeneous. That is, since each ASVM has its own instruction set, the Maté architecture assumes that every node is running the same ASVM. This works fine for simpler deployments, such as habitat monitoring or tracking, but poses problems when a network has several different varieties of mote. For example, the 2003 deployment on Great Duck Island had motes deployed in burrows to monitor birds and motes deployed above ground to monitor weather [102].

Supporting heterogeneous networks would allow Maté ASVMs to be used in these more complex application scenarios. The simplest approach is to deploy different ASVMs. This raises issues with code propagation, however. On one hand, the different kinds of motes should be able to collaboratively operate shared services, such as collection routing. On the other, they cannot necessarily share code. From a handler standpoint, every mote could store all handlers, but only executes the ones relevant to its role in the network. Operations, however, are a bit more complex: if nodes have different sensors, then some sort of platform disambiguation is needed.

7.5.4 Propagation

Following the assumption of homogeneity, Maté assumes that it must disseminate code to every node. The results in Chapter 5 showed that actually disseminating code is inexpensive: selective dissemination would not conserve much energy, as periodic maintenance would still be required. However, selective dissemination could save RAM. As discussed above, the simplest way to support networks with heterogeneous handlers is to disseminate every handler to every node and waste RAM. A much better solution would be to support selective dissemination.

Generally, selective dissemination is simple as long as the dissemination targets are connected. For example, one could imagine adding a simple predicate to Trickle that controls what it considers "new code." However, once the dissemination targets are disconnected, the problem becomes more complex, requiring a spanning tree or other connectivity mechanism. One could also establish "clusters" of connected regions, and use a protocol such as Firecracker to seed those regions [65].

The ability to selectively disseminate code leads to interesting possibilities at the scripting level: there is no reason that code propagation cannot be controlled by Maté programs. For example, code-driven propagation could be used to implement agent programming models [104].

7.5.5 ASVM Synthesis

Currently, building an ASVM is an explicit process: a user specifies the language, handlers and functions. Choosing the right set of extensions poses the same problems and challenges as designing any other programming interface. Chances are that new lengthy deployments will undergo a few iterations of ASVM refinement as users gain experience with the network. Mechanisms that help ASVM builders decide on what extensions to incorporate based on prior use would simplify this iteration. Depending on how tightly the compiler and ASVM are tied, one could have the toolchain generate *superoperators* [85], which are primitives for common code sequences; the compiler compiles these sequences to a single opcode. This sort of optimization/compression is laborious for a user, but could be a simple way for the toolchain to reduce code size, and there are existing automated techniques to do so [85].

Chapter 8

Conclusion

This dissertation proposed using application specific virtual machines (ASVMs) as the underlying runtime for retasking deployed mote networks. The resource constraints, scale, and embedment of mote networks means that a mote runtime must provide three key features to a user: a suitable programming interface, fault tolerance, and efficiency. Depending on the application domain and user preferences, what constitutes a suitable programming interface can vary widely. This means that while a single ASVM can provide all three for an application domain, the diversity of domains means that a single ASVM is insufficient. Developers need a way to easily build ASVMs for a wide range of domains without sacrificing network longevity.

To explore whether ASVMs provide the needed retasking flexibility while remaining energy efficient, be designed and implemented Maté, an architecture for building ASVMs. We built several ASVMs for a range of application domains and compared them to existing runtime solutions. The fact that ASVMs could be used to support several different proposals for programming models while having a negligible energy cost demonstrated the utility and merits of the approach.

The effectiveness of ASVMs leads to several open questions. Foremost among them is the scope and functionality of Maté extensions. One benefit the extensions model has is re-use; as developers write new handlers and operations (either languages or functions), others can take advantage of the functionality. Just as TinyOS has seen a growth in the set of components available to OS programmers, as Maté-based ASVMs

increase in use, future developers will have a greater library to draw from. For example, we are currently working with researchers at UC Davis to include their extensions for agent-based programming.

For sensor networks to be useful, they must be easy to use. Just as early operating systems introduced the notion of user-land protection and processes to simplify application development in traditional systems, Maté proposes ASVMs to simplify application development in sensor networks. Separating the user from the complexities faced by native mote code will make sensor network programming accessible to a much wider audience, and thereby enable many new uses and applications.

Chapter 9

Appendices

9.A SmokeVM and SaviaVM

SmokeVM

```
<VM NAME="SmokeVM"
 DESC="A VM for the SmokeNet project."
 DIR="../../../../apps/SmokeVM">

<SEARCH PATH="../opcodes">
<SEARCH PATH="../contexts">
<SEARCH PATH="../languages">
<SEARCH PATH="../../fire/lib/Flood">
<SEARCH PATH="../../fire/lib/VM">
<SEARCH PATH="../../fire/lib/components">
<SEARCH PATH="../../fire/lib/Health">
<SEARCH PATH="../../fire/lib/Neighborhood">
<SEARCH PATH="../../fire/lib/interfaces">
<SEARCH PATH="../../fire/lib/Path">
<SEARCH PATH="../../fire/lib/types">
<SEARCH PATH="../../fire/hardware">

<LOAD FILE="light_stoplight.vmsf">

<LANGUAGE NAME="tinyscript">

<FUNCTION NAME="bclear">
```

```
<FUNCTION NAME="bfull">
<FUNCTION NAME="bsize">
<FUNCTION NAME="bufsorta">
<FUNCTION NAME="bufsortd">
<FUNCTION NAME="eqtype">
<FUNCTION NAME="err">
<FUNCTION NAME="id">
<FUNCTION NAME="int">
<FUNCTION NAME="led">
<FUNCTION NAME="rand">
<FUNCTION NAME="send">
<FUNCTION NAME="sleep">
<FUNCTION NAME="uart">

<CONTEXT NAME="Timer0">
<CONTEXT NAME="Once">
<CONTEXT NAME="Reboot">

<FUNCTION NAME="getawoln">
<FUNCTION NAME="setawoltimeout">
<FUNCTION NAME="addneighbor">
<FUNCTION NAME="updateneighbor">
<FUNCTION NAME="clearneighbors">
<FUNCTION NAME="clearstate">
<FUNCTION NAME="resethealth">
<FUNCTION NAME="setexithop">
<FUNCTION NAME="bootsys">
<FUNCTION NAME="haltsys">
<FUNCTION NAME="onfire">
<FUNCTION NAME="setisexit">
<FUNCTION NAME="stoplights">
<FUNCTION NAME="sethealthtimer">
```

SaviaVM

```
<VM NAME="SapVM"
    DESC="A virtual machine for the Telos RevB platform
          for Redwood Sap deployment."
    DIR="../../../../apps/SapVM"><SEARCH PATH="../opcodes">
<SEARCH PATH="../contexts">
<SEARCH PATH="../languages">
<SEARCH PATH="../SapVM">
<SEARCH PATH="../../Flash">
<SEARCH PATH="../SapVM/Logger">
```

```
<SEARCH PATH="../SapVM/TimeSync">
<SEARCH PATH="../SapVM/CC2420">

<LOAD FILE="telos.vmsf">
<LANGUAGE NAME="tinyscript">

<FUNCTION NAME="bclear">
<FUNCTION NAME="bfull">
<FUNCTION NAME="bsize">
<FUNCTION NAME="bufsorta">
<FUNCTION NAME="bufsortd">
<FUNCTION NAME="eqtype">
<FUNCTION NAME="err">
<FUNCTION NAME="id">
<FUNCTION NAME="int">
<FUNCTION NAME="led">
<FUNCTION NAME="rand">
<FUNCTION NAME="sendlqi">
<FUNCTION NAME="sleep">
<FUNCTION NAME="uart">

<FUNCTION NAME="fread">
<FUNCTION NAME="fwrite">
<FUNCTION NAME="ferase">
<FUNCTION NAME="getdata">
<FUNCTION NAME="storedata">

<CONTEXT NAME="Reboot">
<CONTEXT NAME="SynchTimer">
```

9.B TinyScript

This appendix comtains the TinyScript grammar as well as the primitives the language compiles to.

9.B.1 TinyScript Grammar

TinyScript-file:

variable-list$_{OPT}$*statement-list*$_{OPT}$

variable-list:

variable

variable-list variable

variable:

`shared` *identifier*

`private` *identifier*

`buffer` *identifier*

statement-list:

statement

statement-list statement

statement:

assignment

control-statement

function-statement

control:

if-statement

for-unconditional

for-conditional

if-statement:

`if` *expression* `then` *then-clause* `end if`

then-clause:

statement-list

statement-list `else` *statement-list*

for-unconditional:

`for` *variable-ref* = *expression* `to` *constant-expression step statement-list* `next` *variable-ref*

for-conditional:

`for` *scalar-ref* = *constant-expression step for-condition statement-list* `next` *scalar-ref*

for-condition:

`until` *expression*

`while` *expression*

constant-expression:

constant

function-statement:

function-call ;

function-call:

name (*parameter-list*$_{OPT}$)

parameter-list:

parameter

parameter-list , *parameter*

parameter:

expression

assignment:

l-value = *expression* ;

l-value:

var-ref

buffer-ref

scalar-ref:

identifer

var-ref:

identifier

buffer-ref:

identifier [ ]

identifier [*expression*]

expression:

function-call

constant-expression

variable-expression

paren-expression

unary-expression

binary-expression

conditional-expression

variable-expression:

buffer-access

variable-access

buffer-access:

identifier [ ]

identifier [*expression*]

variable-access:

identifier

paren-expression:

(*expression*)

unary-expression:

- *expression*

≈ *expression*

NOT *expression*

binary-expression:

expression + *expression*

expression - *expression*

expression * *expression*

expression / *expression*

expression % *expression*

expression & *expression*

expression | *expression*

expression # *expression*

expression AND *expression*

expression XOR *expression*

expression OR *expression*

expression EQV *expression*

expression IMP *expression*

conditional-expression:

expression = *expression*

expression != *expression*

expression >= *expression*

expression > *expression*

expression <= *expression*

expression < *expression*

constant:

`[1-9][0-9]*`

identifier:

`[A-Za-z_][A-Za-z_0-9]*`

9.B.2 TinyScript Primitives

```
<LANGUAGE name="TinyScript" desc="A simple, BASIC-like language.">
<PRIMITIVE opcode="halt">
<PRIMITIVE opcode="bcopy">
<PRIMITIVE opcode="add">
<PRIMITIVE opcode="sub">
<PRIMITIVE opcode="land">
<PRIMITIVE opcode="lor">
<PRIMITIVE opcode="or">
<PRIMITIVE opcode="and">
<PRIMITIVE opcode="not">
<PRIMITIVE opcode="lnot">
<PRIMITIVE opcode="div">
<PRIMITIVE opcode="btail">
<PRIMITIVE opcode="eqv">
<PRIMITIVE opcode="exp">
<PRIMITIVE opcode="imp">
<PRIMITIVE opcode="lxor">
<PRIMITIVE opcode="2pushc10">
<PRIMITIVE opcode="3pushc16">
<PRIMITIVE opcode="2jumps10">
<PRIMITIVE opcode="getlocal3">
<PRIMITIVE opcode="setlocal3">
<PRIMITIVE opcode="unlock">
<PRIMITIVE opcode="punlock">
<PRIMITIVE opcode="bpush3" locks=true>
<PRIMITIVE opcode="getvar4" locks=true>
```

```
<PRIMITIVE opcode="setvar4" locks=true>
<PRIMITIVE opcode="pushc6">
<PRIMITIVE opcode="mod">
<PRIMITIVE opcode="mul">
<PRIMITIVE opcode="bread">
<PRIMITIVE opcode="bwrite">
<PRIMITIVE opcode="pop">
<PRIMITIVE opcode="eq">
<PRIMITIVE opcode="gte">
<PRIMITIVE opcode="gt">
<PRIMITIVE opcode="lt">
<PRIMITIVE opcode="lte">
<PRIMITIVE opcode="neq">
<PRIMITIVE opcode="copy">
<PRIMITIVE opcode="inv">
```

Bibliography

[1] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, J. Deng, and R. Han. MANTIS: System Support for MultimodAl NeTworks of In-situ Sensors. In *2nd ACM International Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2003.

[2] A. Acharya, M. Uysal, and J. Saltz. Active disks: programming model, algorithms and evaluation. In *ASPLOS-VIII: Proceedings of the eighth international conference on Architectural support for programming languages and operating systems*, pages 81–91. ACM Press, 1998.

[3] V. Adve, C. Lattner, M. Brukman, A. Shukla, and B. Gaeke. Llva: A low-level virtual instruction set architecture. In *MICRO 36: Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture*, page 205, Washington, DC, USA, 2003. IEEE Computer Society.

[4] E. Amir, S. McCanne, and R. Katz. An active service framework and its application to real-time multimedi a transcoding. In *SIGCOMM '98: Proceedings of the ACM SIGCOMM '98 conference on Appl ications, technologies, architectures, and protocols for computer communication*, pages 178–189. ACM Press, 1998.

[5] T. Anderson, B. Bershad, E. Lazowska, and H. Levy. Scheduler activations: Effective kernel support for the user-level management of parallelism. *ACM Transactions on Computer Systems*, 10(1):53–79, February 1992.

[6] A. Arora, P. Dutta, S. Bapat, V. Kulathumani, H. Zhang, V. Naik, V. Mittal, H. Cao, M. Demirbas, M. Gouda, Y. Choi, T. Herman, S. Kulkarni, U. Arumugam, M. Nesterenko, A. Vora, and M. Miyashita. A line in the sand: a wireless sensor network for target detection, classification, and tracking. *Computer Networks Journal*, 46(5):605–634, 2004.

[7] P. Bahl and V. N. Padamanabhan. Radar: An in-building rf-based user location and tracking system. In *Proceedings of the 19th IEEE Conference on Computer Communications(INFOCOM 2000)*, 2000.

[8] J. R. Bell. Threaded code. *Communications of the ACM*, 16(6):370–372, 1973.

[9] U. B. Berkeley Manufacturing Institute. Fire: Fire information and rescue equipment. http://fire.me.berkeley.edu/, 2005.

[10] B. Bershad, S. Savage, P. Pardyak, E. G. Sirer, D. Becker, M. Fiuczynski, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP-15)*, 1995.

[11] B. N. Bershad. The increasing irrelevance of IPC performance for microkernel-based operating systems. In *Workshop on Micro-Kernels and Other Kernel Architectures*, pages 205–211, Seattle WA (USA), 1992.

[12] J. Beutel, O. Kasten, and M. Ringwald. Poster abstract: Btnodes – a distributed platform for sensor nodes. In *SenSys '03: Proceedings of the 1st international conference on Embedded networked sensor systems*, pages 292–293, New York, NY, USA, 2003. ACM Press.

[13] D. Bhandarkar. Risc versus cisc: a tale of two chips. *SIGARCH Comput. Archit. News*, 25(1):1–12, 1997.

[14] K. P. Birman, M. Hayden, O. Ozkasap, Z. Xiao, M. Budiu, and Y. Minsky. Bimodal multicast. *ACM Transactions on Computer Systems (TOCS)*, 17(2):41–88, 1999.

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

[16] E. A. Brewer and B. C. Kuszmaul. How to get good performance from the cm-5 data network. In *Proceedings of the 8th International Symposium on Parallel Processing*, pages 858–867, Washington, DC, USA, 1994. IEEE Computer Society.

[17] J. Byers, J. Considine, M. Mitzenmacher, and S. Rost. Informed content delivery across adaptive overlay networks. In *Proceedings of the 2002 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 47–60. ACM Press, 2002.

[18] B. M. Cantrill, M. W. Shapiro, and A. H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of USENIX 2004 Annual Technical Conference General Track*, 2004.

[19] B. Carpenter. Architectural principles of the internet. Internet Network Working Group RFC1958, June 1996.

[20] A. Cerpa, N. Busek, and D. Estrin. Scale: A tool for simple connectivity assessment in lossy environments. Technical Report 0021, Sept. 2003.

[21] A. Cerpa and D. Estrin. Adaptive self-configuring sensor networks topologies. In *Proceedings of the Twenty First International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)*, 2002.

[22] D. D. Clark. The design philosophy of the darpa internet protocols. *SIGCOMM Comput. Commun. Rev.*, 25(1):102–111, 1995.

[23] CNN. Redwoods now part of wireless network.
http://www.cnn.com/2003/TECH/science/08/15/coolsc.redwoods/, 2003.

[24] A. Corporation. 8-bit avr microcontroller with 128k bytes in-system programmable flash: Atmega 103(l).

[25] A. Corporation. 8-bit avr microcontroller with 128k bytes in-system programmable flash: Atmega128, atmega 128l summary.

[26] A. Corporation. 8-bit avr microcontroller with 16k bytes in-system programmable flash: Atmega 163, atmega 163l.

[27] I. Corporation. New computing frontiers – the wireless vineyard. http://www.intel.com/technology/techresearch/research/rs01031.htm, 2003.

[28] F. M. Cuenca-Acuna, C. Peery, R. P. Martin, and T. D. Nguyen. PlanetP: Using Gossiping to Build Content Addressable Peer-to-Peer Information Sharing Communities. Technical Report DCS-TR-487, Department of Computer Science, Rutgers University, Sept. 2002.

[29] D. E. Culler, J. P. Singh, and A. Gupta. *Parallel Computer Architecture: A Hardware/Software Approach*. Morgan Kaufmann, 1998.

[30] A. Demers, D. Greene, C. Hauser, W. Irish, and J. Larson. Epidemic algorithms for replicated database maintenance. In *Proceedings of the sixth annual ACM Symposium on Principles of distributed computing*, pages 1–12. ACM Press, 1987.

[31] C. Diot, B. N. Levine, B. Lyles, H. Kassem, and D. Balensiefen. Deployment issues for the IP multicast service and architecture. *IEEE Network*, 14(1):78–88, / 2000.

[32] P. Dutta. On random event detection in wireless sensor networks. Master's thesis, The Ohio State University, 2004.

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

[34] J. Ernst, W. Evans, C. W. Fraser, T. A. Proebsting, and S. Lucco. Code compression. In *Proceedings of the ACM SIGPLAN 1997 conference on Programming language design and implementation (PLDI '97)*, pages 358–365, New York, NY, USA, 1997. ACM Press.

[35] D. Estrin et al. *Embedded, Everywhere: A Research Agenda for Networked Systems of Embedded Computers*. National Acedemy Press, Washington, DC, USA, 2001.

[36] S. Floyd, V. Jacobson, S. McCanne, C.-G. Liu, and L. Zhang. A reliable multicast framework for light-weight sessions and application level framing. In *Proceedings of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 342–356. ACM Press, 1995.

[37] M.-P. I. Forum. Mpi: A message passing interface standard, version 1.1., 1995.

[38] C. W. Fraser, E. W. Myers, and A. L. Wendt. Analyzing and compressing assembly code. In *Proceedings of the ACM SIGPLAN 1984 Symposium on Compiler Construction*. ACM Press, 1984.

[39] D. Ganesan, B. Krishnamachari, A. Woo, D. Culler, D. Estrin, and S. Wicker. An empirical study of epidemic algorithms in large scale multihop wireless networks. UCLA Computer Science Technical Report UCLA/CSD-TR 02-0013, 2002.

[40] D. Gay and J. Hui. Tep 103: Permanent data storage (flash), 2005.

[41] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *SIGPLAN Conference on Programming Language Design and Implementation (PLDI'03)*, June 2003.

[42] L. Gird, T. Stathopoulos, N. Ramanathan, J. Elson, D. Estrin, E. Osterweil, and T. Schoellhammer. A system for simulation, emulation, and deployment of heterogeneous sensor networks. In *Proceedings of the Second ACM Conferences on Embedded Networked Sensor Systems (SenSys)*, 2004.

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

[44] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification*. Addison-Wesley, Reading, MA, 1996.

[45] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (snack). In *Proceedings of the Second ACM International Conference on Embedded Networked Sensor Systems (Sensys '04)*, 2004.

[46] C.-C. Han, R. K. Rengaswamy, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor networks. In *Proceedings of the Third Internaltiona Conference on Mobile Systems, Appliactions, and Services (Mobisys)*, 2005.

[47] S. Hand, A. Warfield, K. Fraser, E. Kotsovinos, and D. Magenheimer. Are virtual machine monitors microkernels done right? In *Proceedings of the Tenth USENIX Workshop on Hot Topics in Operating Systems (HotOS)*, 2005.

[48] W. R. Heinzelman, J. Kulik, and H. Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 174–185. ACM Press, 1999.

[49] M. Hicks, P. Kakkar, J. T. Moore, C. A. Gunter, and S. Nettles. Plan: A packet language for active networks. In *Proceedings of the International Conference on Functional Programming (ICFP)*, 1998.

[50] M. Hicks, J. T. Moore, D. S. Alexander, C. A. Gunter, and S. Nettles. Planet: An active internetwork. In *Proceedings of IEEE INFOCOM*, 1999.

[51] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000. TinyOS is available at http://webs.cs.berkeley.edu.

[52] J. W. Hui and D. Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[53] F. L. R. III. Experience with the development of a microkernel-based, multi-server operating system. In *Workshop on Hot Topics in Operating Systems*, pages 2–7, 1997.

[54] T. Instruments. Msp430x13x, msp430x14x, msp430x14x1 mixed signal microcontroller.

[55] T. Instruments. Msp430x15x, msp430x16x, msp430x161x mixed signal microcontroller.

[56] C. Intanagonwiwat, R. Govindan, and D. Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Proceedings of the International Conference on Mobile Computing and Networking*, Aug. 2000.

[57] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with zebranet. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, oct 2002.

[58] M. F. Kaashoek, D. R. Engler, G. R. Ganger, H. M. Briceño, R. Hunt, D. Mazières, T. Pinckney, R. Grimm, J. Jannotti, and K. Mackenzie. Application performance and flexibility on Exokernel systems. In *Proceedings of the 16th ACM Symposium on Operating Systems Principles (SOSP '97)*, October 1997.

[59] C. Karlof, N. Sastry, and D. Wagner. TinySec: A link layer security architecture for wireless sensor networks. In *Proceedings of the Second International Conferences on Embedded Network Sensor Systems (SenSys)*, 2004.

[60] C. Karlof and D. Wagner. Secure routing in wireless sensor networks: Attacks and countermeasures. In *IEEE International Workshop on Sensor Network Protocols and PApplications (SNPA)*, may 2003. To appear.

[61] S. Kim. Wireless sensor networks for structural health monitoring. Master's thesis, University of California at Berkeley, 2005.

[62] S. Kim, D. Culler, J. Demmel, G. Fenves, T. Oberheim, and S. Pakzad. Structural health monitoring of the golden gate bridge. http://www.cs.berkeley.edu/ binetude/ggb/, 2005.

[63] D. Kotz, C. Newport, and C. Elliott. The mistaken axioms of wireless-network research. Technical Report TR2003-467, Dept. of Computer Science, Dartmouth College, July 2003.

[64] P. Levis and D. Culler. Maté: a tiny virtual machine for sensor networks. In *Proceedings of the ACM Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS X)*, Oct. 2002.

[65] P. Levis and D. Culler. The firecracker protocol. In *Proceedings of the 11th ACM SIGOPS European Workshop*, 2005.

[66] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Simulating large wireless sensor networks of tinyos motes. In *Proceedings of the First ACM Conference on Embedded Networked Sensor Systems (SenSys 2003)*, 2003.

[67] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, and D. Culler. TinyOS: An operating system for wireless sensor networks. In *Ambient Intelligence*, New York, NY, 2004. Springer-Verlag.

[68] T. Liu and M. Martonosi. Impala: a middleware system for managing autonomic, parallel sensor systems. In *Proceedings of the ninth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 107–118. ACM Press, 2003.

[69] J. Luo, P. Eugster, and J.-P. Hubaux. Route driven gossip: Probabilistic reliable multicast in ad hoc networks. In *Proc. of INFOCOM 2003*, 2003.

[70] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: An acquisitional query processing system for sensor networks. *Transactions on Database Systems (TODS)*, 2005.

[71] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a Tiny AGgregation Service for Ad-Hoc Sensor Networks. In *Proceedings of the ACM Symposium on Operating System Design and Implementation (OSDI)*, Dec. 2002.

[72] E. T. Magazine. Second-generation 1t-sram saves even more power. http://www.electronicstalk.com/news/mos/mos109.html, 2001.

[73] G. Mainland, D. C. Parkes, and M. Welsh. Decentralized, adaptive resource allocation for sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)*, 2005.

[74] M. Maroti, B. Kusy, G. Simon, and A. Ledeczi. The flooding time synchronization protocol. In *Proceedings of the Second ACM International Conference on Embedded Networked Sensor Systems (Sensys '04)*, 2004.

[75] L. W. McVoy and S. R. Kleiman. Extent-like performance from a UNIX file system. In *Proceedings of the USENIX Winter 1991 Technical Conference*, pages 33–43, Dallas, TX, USA, 21–25 1991.

[76] D. Mertz. Yellow dog linux on power mac g5: A linux on power developer's workstation. http://www-106.ibm.com/developerworks/linux/library/l-ydlg5.html?ca=dgr-mw05LinxOnG5, 2004.

[77] S. Microsystems. *Virtual Machine Specification, Java Card Platform, Version 2.2.1*. Sun Microsystems, 2003.

[78] S. Nath, P. B. Gibbons, S. Seshan, and Z. R. Anderson. Synopsis diffusion for robust aggregation in sensor networks. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems*, pages 250–262, New York, NY, USA, 2004. ACM Press.

[79] S.-Y. Ni, Y.-C. Tseng, Y.-S. Chen, and J.-P. Sheu. The broadcast storm problem in a mobile ad hoc network. In *Proceedings of the fifth annual ACM/IEEE international conference on Mobile computing and networking*, pages 151–162. ACM Press, 1999.

[80] B. Nowicki. Rfc 1094. nfs: Network file system protocol specification., 1989.

[81] A. Perrig. The biba one-time signature and broadcast authentication protocol. In *ACM Conference on Computer and Communications Security*, pages 28–37, 2001.

[82] D. S. Platt. *Introducing Microsoft .NET, 2nd ed.* Microsoft Press, 2002.

[83] J. Polastre, R. Szewczyk, and D. Culler. Telos: Enabling ultra-low power wireless research. In *Proceedings of the Fourth International Conference on Information Processing in Sensor Networks: Special track on Platform Tools and Design Methods for Network Embedded Sensors (IPSN/SPOTS)*, 2005.

[84] N. B. Priyantha, A. Miu, H. Balakrishnan, and S. Teller. The Cricket Compass for context-aware mobile applications. In *Proceedings of the 7th ACM MOBICOM*, Rome, Italy, July 2001.

[85] T. A. Proebsting. Optimizing an ansi c interpreter with superoperators. In *POPL '95: Proceedings of the 22nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 322–332, New York, NY, USA, 1995. ACM Press.

[86] R. Rashid, R. Baron, A. Forin, M. J. David Golub, D. Julin, D. Orr, and R. Sanzi. Mach: A foundation for open systems. In *Proceedings of the Second Workshop on Workstation Operating Systems(WWOS2)*.

[87] E. D. Rather and C. H. Moore. The forth approach to operating systems. In *ACM 76: Proceedings of the annual conference*, pages 233–240, New York, NY, USA, 1976. ACM Press.

[88] N. Reijers and K. Langendoen. Efficient code distribution in wireless sensor networks. In *Proceedings of the Second ACM International Workshop on Wireless Sensor Networks and Applications (WSNA '03)*, 2003.

[89] I. RF Monolithics. Tr1000: 916.5 mhz hybrid transceiver, 2003.

[90] P. Rundel and E. Graham. Research project: The effect of rocks on temperature, moisture, and nutrient gradients in arid soils. http://research.cens.ucla.edu, 2005.

[91] J. H. Saltzer, D. P. Reed, and D. D. Clark. End-to-end arguments in system design. *ACM Trans. Comput. Syst.*, 2(4):277–288, 1984.

[92] Y. Sasson, D. Cavin, and A. Schiper. Probabilistic broadcast for flooding in wireless networks. Technical Report IC/2002/54, 2002.

[93] B. Schwartz, A. W. Jackson, W. T. Strayer, W. Zhou, R. D. Rockwell, and C. Partridge. Smart packets: Applying active networks to network management. *ACM Transations on Computer Systems*, 2000.

[94] C. Sharp, S. Schaffert, A. Woo, N. Sastry, C. Karlof, S. Sastry, and D. Culler. Design and implementation of a sensor network system for vehicle tracking and autonomous interception. In *Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[95] N. Shaylor, D. N. Simon, and W. R. Bush. A java virtual machine architecture for very small devices. In *LCTES '03: Proceedings of the 2003 ACM SIGPLAN conference on Language, compiler, and tool for embedded systems*, pages 34–41, New York, NY, USA, 2003. ACM Press.

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

[97] E. G. Sirer, S. McDirmid, and B. N. Bershad. Verifying java verifiers. Talk at Workshop on Security and Languages, 1998.

[98] R. L. Sites, A. Chernoff, M. B. Kirk, M. P. Marks, and S. G. Robinson. Binary translation. *Commun. ACM*, 36(2):69–81, 1993.

[99] T. Stathopoulos, J. Heidemann, and D. Estrin. A remote code update mechanism for wireless sensor networks. Technical Report CENS Technical Report 30, Center for Embedded Networked Sensing, 2003.

[100] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *SIGCOMM '01: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 149–160, New York, NY, USA, 2001. ACM Press.

[101] Sun Microsystems, Inc. The K Virtual Machine (KVM). `http://java.sun.com/products/kvm/`.

[102] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. An analysis of a large scale habitat monitoring application. In *Proceedings of the Second ACM Conference on Embedded Networked Sensor Systems (SenSys 2004)*, 2004.

[103] R. Szewczyk, J. Polastre, A. Mainwaring, and D. Culler. Lessons from a sensor network expedition. In *Proceedings of the First European Workshop on Sensor Networks (EWSN)*, Berlin, Germany, Jan. 2004.

[104] L. Szumel, J. LeBrun, and J. D. Owens. Towards a mobile agent framework for sensor networks. In *Proceedings of the Second IEEE Workshop on Embedded Networked Sensors*, 2005.

[105] D. Tennenhouse and D. Wetherall. Towards an active network architecture. In *Computer Communication Review, 26(2)*, 1996.

[106] R. Tolksdorf. Programming languages for the java virtual machine. http://www.robert-tolksdorf.de/vmlanguages, 2004.

[107] G. Tolle and D. Culler. Design of an application-cooperative management system for wireless sensor networks. In *Proceedings of the Second European Workshop on Wireless Sensor Networks (EWSN)*, 2005.

[108] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active Messages: A Mechanism for Integrated Communication and Computation. In *Proceedings of the International Symposium on Computer Architecture*, pages 256–266, 1992.

[109] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauser. Active messages: a mechanism for integrating communication and computation. In *Proceedings of the 19th Annual International Symposium on Computer Architecture*, pages 256–266, May 1992.

[110] M. Welsh and D. Culler. Virtualization considered harmful: OS design directions for well-conditioned services. In *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS VIII)*, Schloss Elmau, Germany, May 2001.

[111] M. Welsh and G. Mainland. Programming sensor networks with abstract regions. In *First USENIX/ACM Symposium on Network Systems Design and Implementation (NSDI)*, 2004.

[112] D. Wetherall. Active network vision and reality: Lessons from a capsule-based system. In *Proceedings the 17th ACM Symposium on Operating System Principles (SOSP'99)*, 1999.

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

[114] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *Proceedings of the first international conference on Embedded networked sensor systems*, pages 14–27. ACM Press, 2003.

[115] J. Zhao and R. Govindan. Understanding packet delivery performance in dense wireless sensor networks. In *Proceedings of the First International Conference on Embedded Network Sensor Systems*, 2003.