SYSTEM SUPPORT FOR PERPETUAL MOBILE TRACKING

A Dissertation Presented

by

JACOB M. SORBER

Submitted to the Graduate School of the
University of Massachusetts Amherst in partial fulfillment
of the requirements for the degree of

DOCTOR OF PHILOSOPHY

September 2010

Computer Science

© Copyright by Jacob M. Sorber 2010

All Rights Reserved

SYSTEM SUPPORT FOR PERPETUAL MOBILE TRACKING

A Dissertation Presented

by

JACOB M. SORBER

Approved as to style and content by:

____

Mark D. Corner, Chair

____

Emery D. Berger, Member

____

Deepak Ganesan, Member

____

Brian N. Levine, Member

____

Tilman Wolf, Member

____

Andrew Barto, Department Chair
Computer Science

to Trisha, Maren, Eliza, and Jane

ACKNOWLEDGEMENTS

This dissertation is the product of the time, ideas, advice, support, and hard work of many people. I am indebted to them all. Specifically, I want to acknowledge:

Mark Corner, my advisor, who has funded my research since 2004. A patient mentor through the many failures, frustrations, and victories, I would not be here without Mark's advice and encouragment along the way.

Professors Emery Berger, Brian Levine, Deepak Ganesan, and Tilman Wolf who served on my committee and whose advice and encouragement on research, career planning, and job hunting has been invaluable.

My wife, Trisha, whose selfless love has made graduate school infinitely more enjoyable; and my daughters, Maren, Eliza, and Jane, who always think my research is cool and who happily provide diversions when I need a break.

The excellent members (past and present) of the PRISMS lab, especially Nilanjan Banerjee, Aruna Balasubramanian, Ben Ransford, Shane Clark, Matt Brennan, Alex Kostadinov, and Matt Garber, with whom I have worked on several different research projects. Also, Allison Clayton for recording lectures for me during chemotherapy in 2004.

My parents and brothers who have supported me throughout my life and have never discouraged a crazy idea. My grandfather, Glen Nelson, whose positive outlook and encouragement have been a great source of help to me and my family.

Matt Gruwell, Mark and John Hyde, Paul Picard, Harry Remer, and others for flyfishing trips, bike rides, and other diversions.

All of my other family and friends for the many ways they have made graduate school and life much more enjoyable. It would be impossible to list them all.

ABSTRACT

SYSTEM SUPPORT FOR PERPETUAL MOBILE TRACKING

SEPTEMBER 2010

JACOB M. SORBER

B.Sc., BRIGHAM YOUNG UNIVERSITY

M.Sc., UNIVERSITY OF MASSACHUSETTS, AMHERST

Ph.D., UNIVERSITY OF MASSACHUSETTS AMHERST

Directed by: Professor Mark D. Corner

Recent advances in low-power electronics, energy harvesting, and sensor technologies are poised to revolutionize mobile and embedded computing, by enabling networks of mobile sensor devices that are long-lived and self-managing. When realized, this new generation of *perpetual systems* will have a far-reaching and transformative impact, improving scientists' ability to observe natural phenomena, and enabling many ubiquitous computing applications for which regular maintenance is not feasible.

In spite of these benefits, perpetual systems face many programming and deployment challenges. Conditions at runtime are unknown and highly variable. Variations in harvested energy and energy consumption, as well as mobility-induced changes in network connectivity and bandwidth require systems that are able to adapt gracefully at run-time to meet different circumstances. However, when programmers muddle adaptation details with application logic, the resulting code is often difficult to both understand and maintain.

Relying on system designers to correctly reason about energy fluctuations and effectively harness opportunities for cooperation among mobile nodes, is not a viable solution.

This dissertation demonstrates that perpetual systems can be designed and deployed without sacrificing programming simplicity. We address the challenges of perpetual operation and energy-aware data delivery in the context of several applications, including in situ wildlife tracking and vehicular networks. Specifically, we focus on two specific systems. Eon, the first energy-aware programming language, allows programmers to simply express application specific energy policies and then delegate the complexities of energy-aware adaptation to the underlying system. Eon automatically manages application energy in order to indefinitely extend a device's operating lifetime, requiring only simple annotations from the programmer. The second system, Tula, is a system that automatically balances the inherently dependent activities of data collection and data delivery, while also ensuring that devices have fair access to network resources. In our experiments, Tula performs within 75% of the optimal max-min fair rate allocation.

TABLE OF CONTENTS

Page

ACKNOWLEDGEMENTS . . . . **v**

ABSTRACT . . . . **vi**

LIST OF TABLES . . . . **xii**

LIST OF FIGURES . . . . **xiii**

CHAPTER

1. INTRODUCTION **1**

1.1 Thesis Statement 2
1.2 Dissertation Overview 2

2. PERPETUAL SYSTEMS **4**

2.1 Energy Awareness 6

2.1.1 Modeling Energy 7
2.1.2 Measuring Energy 7

2.2 Adapting to Changing Conditions 9
2.3 Sparse Mobile Networks 10

3. APPLICATIONS AND DEPLOYMENTS: MOBILE TRACKING **12**

3.1 Wildlife Tracking 12

3.1.1 TurtleNet 14
3.1.2 Mongooses 15
3.1.3 Discussion 16

3.2 UMass DieselNet 17

4. EON: LANGUAGE AND RUNTIME SUPPORT FOR PERPETUAL SYSTEMS . . . 18

4.1 The Eon Programming Language . . . 20

4.1.1 Basic Eon Syntax . . . 21
4.1.2 Eon Extensions . . . 24

4.1.2.1 Power states . . . 24
4.1.2.2 Adaptive Timers . . . 25
4.1.2.3 Energy-State Based Paths . . . 26
4.1.2.4 Implementing Concrete Nodes . . . 26
4.1.2.5 Discussion . . . 27

4.2 The Eon Runtime System . . . 28

4.2.1 Design Goals . . . 28
4.2.2 Energy Adaptation Algorithm . . . 29

4.2.2.1 Energy Attribution and Consumption . . . 30
4.2.2.2 Energy Source Model . . . 31

4.3 Implementation and Deployment . . . 32

4.3.1 Software . . . 32

4.3.1.1 Compiler . . . 32
4.3.1.2 Runtime System . . . 33
4.3.1.3 Trace-Based Simulator . . . 33

4.3.2 Hardware . . . 33

4.4 Deployment . . . 34

4.4.1 Turtle Tracking . . . 35
4.4.2 Automobile Tracking . . . 36
4.4.3 Remote Camera . . . 37

4.5 Evaluation . . . 37

4.5.1 User Study . . . 37
4.5.2 Adaptation . . . 41
4.5.3 Impact of Energy-State Based Paths . . . 43
4.5.4 System Overhead . . . 45
4.5.5 Measurement Accuracy . . . 46
4.5.6 Impact of Battery Capacity . . . 47

4.6 Related Work 48
4.7 Discussion 50

5. TULA: FAIR AND BALANCED DATA DISSEMINATION 51

5.1 Challenges for Perpetual Networks 53

5.1.1 Challenges 55
5.1.2 Design Goals 56

5.2 Tula Architecture 58

5.2.1 Adapting Sensing 59
5.2.2 Adapting Routing 60

5.3 Rate Allocation 60

5.3.1 Objective function 62
5.3.2 Energy conservation constraint 62
5.3.3 Downstream constraint 63
5.3.4 Upstream constraint 63

5.4 Incorporating Routing 64

5.4.1 Routing through multiple nodes 64
5.4.2 Replication 67
5.4.3 Transitive routing 67
5.4.4 Routing through an upstream neighbor 68

5.5 Implementation 69

5.5.1 NesC implementation 69
5.5.2 Trace-based simulator 70

5.6 Evaluation 70

5.6.1 Methodology 70

5.6.1.1 Trace collection 72
5.6.1.2 Optimal rate allocation using an oracle 73

5.6.2 Network Performance 73

5.6.2.1 Static rate allocation policies 73
5.6.2.2 Semi-adaptive rate allocation policies 74
5.6.2.3 Network performance over DieselNet and Mesh 75

5.6.3 Fairness 76
5.6.4 Overhead 77

5.7 Related Work 78

5.7.1 Mobile sensor networks 78
5.7.2 Low power sensor networks 79
5.7.3 Challenged networks 79
5.7.4 Fair network rate allocation 79

5.8 Discussion 80

6. CONCLUSIONS AND FUTURE WORK 81

6.1 Conclusion 81
6.2 Future Work 82
6.3 The Final Word 83

BIBLIOGRAPHY 84

LIST OF TABLES

Table **Page**

4.1 Measurements of Eon overhead in comparison to GPS readings. 46

5.1 List of inputs that are exchanged between n and its neighbors to solve the COP. Variables marked *(up)* are exchanged from n's upstream neighbors, and variables marked *(down)* are exchanged with n's downstream neighbors. 64

5.2 Variables that are estimated locally by n to solve the COP 64

5.3 Energy to sense vs. send for common sensors, and the XE1205 low-power radio 71

5.4 Measurements of Tula overhead. 78

LIST OF FIGURES

Figure **Page**

2.1 Energy traces from solar-powered GPS devices over a two week period. 5

3.1 An early TurtleNet test deployment on a snapping turtle. 13

3.2 A gopher tortoise equipped with a TurtleNet tracking device. 13

3.3 A small indian mongoose (Herpestes auropunctatus) equipped with a tracking device. 15

4.1 A condensed version of Eon source code for the tracking application used in TurtleNet 22

4.2 A graphical representation of the simplified turtle tracking code, shown in Figure 4.1 23

4.3 Sample State Order. 25

4.4 The two implementations of the energy measurement and charging board with a Mica2Dot and a TinyNode. 34

4.5 Photos of two of the test applications, a turtle tracking device, and a remote camera. 35

4.6 User study results 40

4.7 The average number of daily GPS readings taken are shown for different energy policies and energy traces. Despite large variations in energy supply, Eon is able to accurately approximate the best sustainable energy policy. 43

4.8 This figure shows the amount of each trace's energy that is consumed by different parts of the system. The percent dead time is also shown for traces that are not sustainable, above the corresponding bar. 44

4.9 Frame rates for a remote camera application are shown over a 16-month trace, comparing Eon to two static policies. Periods of time when the Eon camera disabled streaming as well as periods of dead time for the static policies are shown across the bottom. 45

4.10 Device dead time is shown for different battery sizes for systems using one and three solar panels. Performance using Eon's EWMA predictor is compared with perfect energy prediction (Oracle). The benefit of better energy prediction is most notable when using a very small battery and the cost of prediction errors is greatest. 48

5.1 Daily solar energy is shown for a TurtleNet node before and after hibernation. 53

5.2 The average daily energy harvested by each TurtleNet node during a 1-month trace. 54

5.3 Harvested energy plotted against number of meetings for each node. Energy-rich nodes are not necessarily better connected and vice versa. 55

5.4 CDF of the pair-wise meeting frequency during 1 month of TurtleNet operation. While some meetings occur too infrequently to be very useful, 50% of the node pairs repeat at least 5 times. 57

5.5 The Tula architecture. 58

5.6 A simplified example to illustrate the Tula distributed allocation algorithm. The algorithm is executed by node n, whose upstream neighbors are $u_1, u_2, \dots u_k$ 61

5.7 Energy allocation problem formulation solved by node n. The goal is to estimate r_n, the local sensing rate and r_i, the rate at which n can route packets for each of its neighbors u_i 65

5.8 Scenarios that complicate the simple Tula allocation algorithm 66

5.9 Comparison of three static allocation policies, Tula and Optimal. The policies are compared across three metrics: battery dead time, energy wasted since the battery was full and could not charge, and average delivery rate. Tula avoids dead time and wasted energy successfully, and delivers within 92% of the oracle-based optimal policy. 71

5.10 Comparison of two semi-adaptive allocation polices, Tula and Optimal. The comparison is performed for different sensor applications with varying sensing to routing ratio. 72

5.11 Delivery rate of Tula normalized to the optimal delivery rate over three networks configurations: TurtleNet, a static 4x4 grid mesh network, and the DieselNet vehicular traces. 76

5.12 TurtleNet traces: Average per-node delivery rate 77

5.13 DieselNet traces: Average per-node delivery rate 77

5.14 Mesh: Average per-node delivery rate 77

CHAPTER 1

INTRODUCTION

Due to three key innovations—small programmable sensors, energy harvesting [41,74], and disruption tolerant networking—mobile systems are poised to answer many questions about a wide range of natural and manmade systems. Recent efforts focusing on zebras [88], whales [25], turtles [74], people [37], and vehicles [20] have shown that in situ monitoring using embedded devices can provide unprecedented and transformational data. When these systems harvest energy from their environment and gather data in a robust manner, they can become *perpetual* and self managing, streaming data directly to scientists for decades.

Unfortunately, current mobile software systems have lagged behind. In particular, many biological systems are inherently sparse and mobile and require longitudinal, multi-sensor deployments, something that is poorly supported by current energy management systems. Traditional energy management in embedded systems and applications has largely neglected the challenges of two critical factors: *perpetual operation*, a paradigm by which nodes are assumed to operate forever; and *mobility*, which creates variations in network connectivity and bandwidth as well as energy consumption and production. Node mobility, unpredictable network connectivity, and uncertain energy availability represent the greatest challenges for untethered systems.

Expecting programmers to understand and correctly address these dynamic conditions at run time is unreasonable. Using standard programming languages and tools to build systems that adapt to shifting conditions, system designers are forced to incorporate adap-

tation with the core logic of the system. Such programs are difficult to port, maintain, and understand, even for experts.

1.1 Thesis Statement

In response to the opportunities and challenges presented by perpetual systems, this dissertation seeks to establish the following thesis:

Untethered mobile systems can *operate perpetually* and *deliver data effectively*, without placing undue burden on system designers.

1.2 Dissertation Overview

In support of this thesis, we present novel programming language, runtime system, and network techniques that allow system designers to simply build mobile tracking systems that are energy-aware and self-managing. The organization of this dissertation is as follows.

Chapter 2 describes the benefits of perpetual systems, as well as the challenges preventing their deployment, including variation and uncertainty in both energy harvesting and energy consumption. We outline the fundamental requirements for perpetual mobile systems—namely, making mobile devices energy-aware, adapting system behavior in response to energy information, and communication in the presence of dynamically changing network conditions.

We describe, in Chapter 3, the focus application for this dissertation—mobile tracking—as well as the deployments and mobile networks that we use to both motivate and evaluate our work. The primary example is TurtleNet [74], a network of 17 mobile tracking devices that we deployed in 2008 on endangered tortoises in collaboration with Biologists at the University of Southern Mississippi. We also describe another study focused on invasive mongooses, as well as UMass DieselNet, a vehicular network including more than 40 transit busses servicing the Amherst, MA area. These systems contribute motivation and useful data traces that we have used in this work.

Chapter 4 describes Eon a language and runtime system that simplifies the writing, debugging, and maintaining of adaptive programs. Eon explicitly exposes the structure of a program's data and control flow, and provides a simple way to associate specific program behavior with energy preferences. The Eon runtime system uses this information to measure energy harvesting and consumption and automatically adjust program behavior as energy conditions change.

Chapter 5 describes Tula, a system that extends Eon to account for network cooperation in addition to managing local energy resources. Tula brings together adaptive sensing (e.g. Eon [74]) and disruption tolerant networking [5, 75] in order to balance the inherently dependent activities of sensing (data collection) and data delivery. Tula also ensures that devices have fair access to network resources.

Finally, Chapter 6 describes our conclusions and future work.

CHAPTER 2

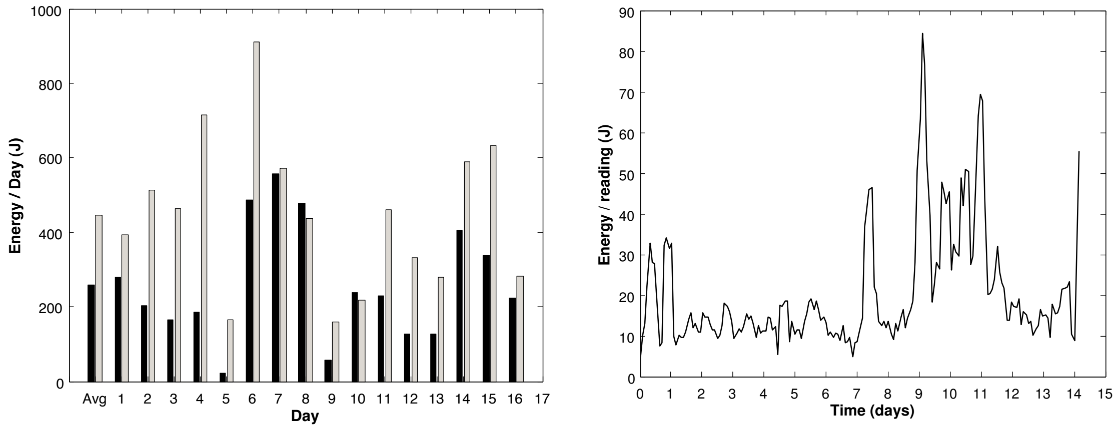
PERPETUAL SYSTEMS

Research on mobile computing systems has traditionally operated under one of two assumptions: (1) that every mobile device will eventually run out of energy, or (2) that every device has access to a wired energy source, such as a vehicle's battery or a user who ensures that the device's battery is periodically recharged. These assumptions, while appropriate for vehicular networks [6,20], laptops, and mobile phones, encourage designers to build systems that depend heavily on either infrastructure or frequent user maintenance.

Many proposed applications and systems—including pervasive computing applications with tens-to-hundreds of devices per user [62] and networks of sensors collecting scientific data [25, 74, 83] in remote locations—violate these assumptions. For these systems, success requires that devices operate for long periods of time without wired infrastructure and without regular maintenance. By harvesting solar, wind, or vibration energy [49, 59, 69] from the environment, it is possible for a mobile device to operate *perpetually*, limited only by the eventual decay of its hardware components. Some ideal applications for *perpetual systems* include wildlife tracking (TurtleNet [74], ZebraNet [88]), volcanic eruption monitoring [83], and forest fire detection [36, 58].

However, despite their deployment advantages, these *perpetual systems* face numerous challenges.

- **Dynamic energy availability.** Mobile devices that rely on environmental energy must cope with changing energy conditions. Harvested energy is often difficult to predict, and may change dramatically with location, time of day, time of year, weather, and other environmental factors. An example of this variation is shown in



(a) Daily solar energy harvested by two devices.

(b) The amount of energy needed to take a GPS reading over the same period.

Figure 2.1. Energy traces from solar-powered GPS devices over a two week period.

Figure 2.1(a). The amount of energy gathered by two mobile, solar-powered devices over the same two-week period is plotted on a per-day basis. Although both devices show elements of the same general weather trend, the two devices show significant variation in the amount of gathered energy.

- **Varying energy costs.** The amount of energy required to perform tasks also may vary widely, making it difficult to plan for future energy needs. Figure 2.1(b) shows the amount of energy that one device required to acquire individual GPS readings over the same two week period as Figure 2.1(a). Because the device is mobile and cloud cover, foliage, and terrain vary, the amount of time (and thus energy) required to synchronize with satellites varies over an order of magnitude. Comparing the graphs from Figures 2.1(a) and 2.1(b) shows that times of substantial cost do not necessarily correspond with times of plentiful energy; in this example, they are nearly opposite.

- **Uncertain network conditions.** Perpetual systems are often deployed without any prior knowledge about future network conditions. A device may never have a direct wireless connection to the network, and must rely on its neighbors' mobility

for communication. Without basic parameters like connectivity, bandwidth, energy availability in the network, it is impossible to appropriately tune routing and power-management.

In order to cope with shifting energy conditions and changing topologies, perpetual systems *must* adapt their behavior as conditions change. All system tasks—data processing, communication, sensing, etc—consume precious energy, and consuming too much energy will exhaust energy reserves, leaving devices unable to provide desired functionality. Conversely, an overly conservative approach wastes resources and needlessly sacrifices system performance. In order to avoid both of these extremes, perpetual systems must be able to (1) reason about energy conditions, (2) trade application fidelity for energy savings, and (3) communicate over sparse and changing connectivity.

2.1 Energy Awareness

Historically, computing systems have ignored energy as much as possible. A desktop computer ignores energy completely. In response to limited battery capacity, a laptop, PDA, or mobile phone estimates the energy stored in its battery, notifying the user when more energy is needed. In this latter case, the *user* is energy-aware and assumes responsibility for energy management, allowing the device's operating system and programs to remain energy-oblivious.

However, increases in scale and distance from users, reductions in size and energy storage, and a desire for predictable system lifetimes are expanding the role of energy in untethered computing systems. Recent systems have tuned network protocols [33, 72] and automatically adjusted sensor duty cycles [44] based on energy information provided by a variety of models and measurement techniques.

2.1.1 Modeling Energy

Using software models to estimate energy consumption is an attractive solution, since most current hardware platforms are still unable to measure their own energy consumption. The ECOSystem [87] estimated a laptop's energy consumption based on the utilization of the CPU, disk, and network. Dunkels et al. [17] similarly provided a linear model for mote-class sensor devices which uses information about CPU states, communication, and various sensor components to estimate consumption. Similar energy accounting functionality is also provided by a several sensor network simulators (e.g. PowerTOSSIM [73], AEON [48]).

With these tools and models enable system designers to accurately reason about the energy-related implications of their decisions with very little system overhead. Software-only approaches can also be deployed without special hardware support; however, software-only approaches also suffer from two serious limitations. 1) Every time a new hardware component is introduced, energy measurements must be collected in all of its power states and the model must be updated—a tedious and often error-prone process. 2) As mentioned previously, energy consumption can vary due to environmental conditions like temperature, foliage cover, device orientation, satellite configuration, and battery voltage. As a result, systems that estimate energy consumption using only software models may perform poorly when deployed different then those found in the lab.

2.1.2 Measuring Energy

When software-only models prove inadequate, several approaches have been proposed for measuring a device's energy profile. Many recent platforms provide information about battery fullness either by directly measuring the battery voltage or using more sophisticated hardware techniques. Measuring a platform's battery voltage is fast, consumes very little energy, and is supported internally, by many microcontrollers; however, variations due to temperature, chemistry, and battery aging as well as the nonlinear mapping between

voltage and stored energy make accurate estimation difficult. Better fullness estimates can be achieved using a *fuel gauge IC*—commonly found in laptop battery packs—which measures energy consumed over time and compensates for temperature fluctuations, aging, and battery chemistry.

In addition to estimating battery fullness, several approaches for fine-grain energy accounting for battery-powered systems have also been explored. PowerMeasure [52] uses a laptop's built-in fuel gauge chip to try and discriminate the power consumed by various components—a method also used in the Nemesis operating system [63], to discover per-process power consumption. Later, the PowerScope [26] used an external oscilloscope to measure individual process power consumption, as well as the energy consumption of blocks of code using annotations. Additionally, systems like SPOT [40] and iCount [18] provide energy accounting solutions for low-power sensors. Both allow systems to estimate per-process or per-task energy consumption, at a very small cost in extra hardware and energy consumption. The simplest of the two, iCount, requires only a single wire (or board trace) from a switching power supply (already included in most low power sensor devices) to one of the microcontrollers counter pins.

In addition to measuring battery fullness and energy consumption, perpetual systems must also account for energy harvesting. Harvested solar [49], wind [69], or vibration [59] energy from the environment depends on weather patterns and device mobility, and cannot be predicted with only a software model. Algorithms for predicting solar energy have been proposed [43, 44] that use information about past harvesting, time of day, etc. to reduce the uncertainty, however, these algorithms all depend on the ability to measure a device's energy harvesting.

When a mobile device is able to measure its energy reserves (e.g., battery fullness), its energy consumption, and its energy harvesting, it can be deployed into unknown environments without exhaustive measurement and profiling in the lab. Designers can change sensors, solar panels, or switch harvesting technologies without changing their applica-

tions. Section 4.3.2 describes the hardware techniques we have used in this dissertation for measuring energy harvesting and consumption, allowing software to accurately estimate the energy cost of individual program tasks at run-time. By providing a clear picture of a device's energy budget and relating its behaviors to energy consumption, this information makes it possible to accurately adapt as conditions change.

2.2 Adapting to Changing Conditions

Energy awareness is key to the success of any perpetual system, but systems must also make effective use of that energy information. System designers can trade performance and fidelity for energy savings by adjusting system execution in a variety of ways—using low power CPU modes [14], reducing the duty cycle of power-hungry components [68], adjusting the transmit power of wireless radios [84], executing alternative low-power algorithms [65], and disabling optional low-priority functionality [74]; however, employing these adaptation techniques effectively is often challenging, even in simple systems.

One reason for the difficulty is that most systems have multiple significant energy consumers, rather than a single dominant one. For example, a device may have sensors (e.g., accelerometer, magnetometer, or GPS receiver) to collect data, a CPU to process that data, and a radio to deliver the processed results. Sensing, processing, and networking tasks all add to the system's energy load, and no single energy management approach (e.g., CPU modes, sensing rates, or radio transmit power) is likely to extract the energy savings necessary to ensure perpetual operation. In order to achieve the systems largest dynamic power range, multiple techniques must be used in concert.

Dependences between individual energy management techniques complicate matters further. Gathering sensor data at a lower rate also reduces the amount of data that must be transmitted over the radio—thus reducing the energy consumption of both the radio and sensors. Conversely, a simple data compression algorithm may require less CPU energy

than a more complex algorithm, while the lower compression ratio means more data to transmit—potentially increasing the system's energy load.

Adaptation decisions made without application-specific information may also result in behavior that is not useful. For example, sensor data gathered too infrequently may make it impossible to detect interesting events. Data gathered too often, may also provide no benefit while consuming energy that could be spent elsewhere. Some sensors or types of information may be more interesting or valuable than others. Prioritizing them also requires an understanding of the application and its goals.

Finally, the greatest challenge for those designing adaptive systems is that traditional programming languages and operating systems force designers to incorporate all of these adaptation policies, constraints, and algorithms with the core logic of the system. This muddling of system priorities results in programs that are difficult to port, maintain, and understand. There is also a great deal of runtime functionality that must be replicated for each new perpetual system, and every time the system is deployed on a new hardware platform with new energy characteristics. In Chapter 4 describes Eon, a language and runtime system that simplifies the programming of perpetual systems.

2.3 Sparse Mobile Networks

The ability to communicate and deliver data is a critical requirement of nearly all perpetual systems, and traditional networking approaches do not account for the uncertain connectivity, bandwidth, and energy resources of perpetual systems.

Prior work on perpetual systems either use local energy adaptation techniques without considering data delivery [53, 74] or use adaptation techniques for purely static networks [24]. However, adapting to both energy and network variations is considerably more difficult. Collecting sensor readings and delivering those readings costs energy; however, sensing and delivery are inherently dependent and must be balanced if we are to gather as much data from each node as the system's limited energy and bandwidth resources will

permit. Sensing more data than can be delivered by the network is not useful, while gathering less underutilizes the system's potential. Similarly, systems that depend on cooperative, replicating routing protocols [5, 75], must balance the energy devoted to sensing and routing their own data, with energy used to route data for other nodes. Techniques for balancing these various networking concerns are described in Chapter 5.

CHAPTER 3

APPLICATIONS AND DEPLOYMENTS: MOBILE TRACKING

Perpetual systems—those that operate perpetually using harvested energy—represent a new class of mobile system that promises to enable a wide and largely unexplored range of potential applications. This vision includes systems that promise transformational advances by monitoring the movements of animals [25, 74, 88], vehicles [6, 10, 20], and people [37]. Whether the goal is to advance scientific understanding, improve road maintenance, provide ubiquitous connectivity, or effectively share media within social networks, in situ mobile tracking systems involve longitudinal deployments, size constraints, small batteries, and mobile disruption-prone networks—the core challenges of any perpetual system.

This dissertation focuses specifically on addressing the challenges of these perpetual mobile tracking systems. The performance and behavior of such systems depends heavily on dynamic conditions that are difficult to predict. As a result, we inform our technical solutions and test our hypotheses, within the context of several mobile systems—focusing on Turtles, Mongooses, and transit buses—developed and deployed as part of the UMass DOME (Diverse Outdoor Mobile Environment) testbed [1]. This chapter describes the details of these experimental deployments that are relevant to our thesis.

3.1 Wildlife Tracking

Our understanding of the natural world is heavily dependent on gathering data about habitat usage, social behavior, long-term population trends, and movement of species. Unfortunately, In spite of decades-worth of study, the movements and behaviors of most animal species in the wild are completely unknown. Scientists studying animal behavior cur-

Figure 3.1. An early TurtleNet test deployment on a snapping turtle.

Figure 3.2. A gopher tortoise equipped with a TurtleNet tracking device.

rently rely almost exclusively on manual observation and trapping, a highly labor-intensive and low-fidelity process. Mobile untethered systems hold great promise for addressing this challenge by providing unprecedented data resolution and quantity, and by minimizing human influence on test populations. Rather than manually tracking a small number of animals, large numbers of perpetual tracking devices will automatically stream data to scientists for years. This shift promises to answer long-debated questions about habitat usage, population trends, and complex interactions between different species, including humans.

The ZebraNet project [88] provided an initial proof-of-concept for in situ mobility tracking focusing on tracking a small number of Zebras in central Kenya. These first devices were large (> 0.5kg) and masked energy variations using large batteries and solar panels—too large for most animals to carry. Similarly, tracking projects using satellite tags, like WhaleNet [25] have also shown impressive results for animals large enough to carry the satellite tags. While not useful for all but the largest animal species, these initial efforts have set the stage for future mobile tracking systems.

In light of these potential benefits and remaining challenges, we are working with biologists to fundamentally change the way wildlife tracking studies are done—making in situ monitoring a viable option for animals both small and large. To date, this cooperation has resulted in a series of system deployments on threatened tortoises and an anticipated deployment on invasive mongooses.

3.1.1 TurtleNet

TurtleNet is a mobile network affiliated with the UMass DOME [1] testbed and deployed with the goal of overcoming many of the challenges faced by perpetual sensing systems. TurtleNet is a collaborative effort, involving biologists from the University of Massachusetts Amherst and the University of Southern Mississippi and studying several different species of turtles and tortoise.

Many turtle and tortoise species are currently in decline due to roadway death, poaching, and the development of habitat [21]. However, current conservation efforts are hampered by a general lack of detailed movement data. Current tracking methods involve manually tracking the animals using radio telemetry and are limited to a single location fix every 2–3 days for each animal being studied. The turtles often travel more than 1 km between fixes and practical concerns preclude the collection of data at night.

We initially conducted small-scale test deployments on two species of turtles: Wood Turtles (*Clemys insculpta*) and Common Snapping Turtles (*Chelydra serpentina*). Subsequent larger scale deployments on Gopher Tortoises (*Gopherus polyphemus*) began in Fall 2008. Photos of two of these deployments are shown in Figures 3.1 and 3.2. These various tests and deployments have allowed us to explore packaging and attachment methods, evaluate new hardware and runtime systems, and to gain experience in working with biologists and animals.

Our current deployment consists of 17 tracking devices attached to tortoises. Each device consists of a Shockfish TinyNode (processor, flash storage, and low-power radio), a solar panel, a battery, multiple sensors, and additional energy measurement hardware. During operation, the devices record connection opportunities with neighboring nodes and periodic sensor readings, including temperature, GPS coordinates, battery level, solar energy harvested and energy consumption.

Unlike traditional networks, these nodes rarely have an end-to-end connection to one of the two deployed GPRS-enabled base stations, and devices must opportunistically deliver

Figure 3.3. A small indian mongoose (Herpestes auropunctatus) equipped with a tracking device.

collected data using mobile-to-mobile routing [5, 76]. When two mobile nodes are within communication range, called a connection opportunity, they exchange data. This data is stored and then forwarded during subsequent connection opportunities until it is eventually delivered to the sink.

3.1.2 Mongooses

In addition to preserving animal habitats, biologists are also concerned with the management of invasive species. The small Indian mongoose (*Herpestes auropunctatus*), shown in Figure 3.3, was introduced to the island of St. Croix in the late 19th century to control rats. They are presently considered to be a nuisance as well as a possible threat to local fauna, including endangered sea turtle nesting sites, based on tracks found near exhumed nests [61]. Today, biologists study wild mongooses by trapping them and implanting small RFID PIT tags under their skin, so that each individual can be identified when recaptured later. This approach provides only a very coarse view of both movement and behavior, which is inadequate to determine whether mongooses pose a significant threat to indigenous species, or how best to mitigate their impact.

We are currently in the early-stages of another collaboration with biologists at UMass Amherst and Westfield State College focused on addressing these concerns. The biological goals of this project are similar to TurtleNet; however, differences in animal mobility (faster and smaller animals) as well as energy harvesting performance (mongooses don't bask) provides additional challenges requiring different techniques.

3.1.3 Discussion

In both TurtleNet and the mongoose study, the use of small mobile embedded devices has the potential to provide more detailed data and enable researchers to manage deployments involving larger numbers of animals. It is important to note, however, that this dissertation is not focused on biological questions specifically—all animal care matters have been handled by biologists—but rather on the systems design issues raised by working with real systems.

Specifically, both studies face a variety of challenges common to many untethered mobile systems, including size constraints, energy variations, and highly nonuniform network conditions.

Size and weight limitations are inherent to wildlife tracking applications. Animal safety guidelines limit devices to no more than 5% of the animal's weight—less than 50g in our deployment scenarios. Additionally, devices must not interfere with normal movement. For example, tortoises are not able to adjust to unexpected increases in carapace height, potentially resulting in problems navigating tight spaces. While these restrictions clearly preclude the use of heavy or bulky components, they also critically limit energy storage capacity. Since large deployments in remote locations make frequent battery changes or other maintenance unmanageable, such systems must conserve energy in favor of extended system lifetimes.

3.2 UMass DieselNet

In addition to wildlife tracking application, this dissertation also benefits from UMass DieselNet—a mobile network of 40 transit busses—which has provided service to Amherst, MA and the surrounding communities since 2004. In DieselNet, participating busses are each equipped with a small server (Hacom OpenBrick) running Ubuntu Linux, a GPS receiver, and multiple radio interfaces (WiFi, 3G, and 900MHz Digi XTend). Commuters can connect to the server over WiFi, which provides Internet connectivity through the AT&T 3G network. Users can also use an on-line bus locator to determine whether their bus is on schedule. Finally, DieselNet provides a testbed and a large body of mobility and connectivity data that has enabled a wide range of research on disruption-tolerant and heterogenous networks [5, 9, 11, 89, 90]

For the purposes on this dissertation, DieselNet provides a valuable contrast to TurtleNet, with faster-moving nodes, shorter connection events, regular mobility patterns, and greater bandwidth. We use the collected mobility and connectivity data from DieselNet in our design and experiments, in order to provide more generally applicable results. We discuss these experiments in further detail in Section 5.6.

CHAPTER 4

EON: LANGUAGE AND RUNTIME SUPPORT FOR PERPETUAL SYSTEMS

Throughout our experience in building deployable perpetual systems we have repeatedly found that the greatest impediment is programming adaptive systems. Designing adaptive code using traditional tools leads programmers to muddle adaptation details with the core logic of the system—resulting in programs that are difficult to port, maintain, and understand.

For instance, when porting programs between platforms with different energy characteristics, large sections of code must be changed to compensate for increases or decreases in energy consumption and production. Worse yet, incorporating the resource management preferences of scientists necessarily requires a deep understanding of the rest of the program, an untenable solution.

In this chapter, we describe techniques that significantly ease the burden placed on programmers during the development of perpetual mobile systems while opening the tuning of the resource management system to less technical users, like scientists. Specifically, this chapter describes **Eon**, a new language and runtime system designed for programming perpetual computing systems. To our knowledge, Eon is the first *energy-aware* programming language. Eon is a declarative coordination language, based on the Flux language [12], that allows programmers to build programs from code written in a variety of languages, including nesC and C. Eon provides a simple way to associate particular control flows with abstract *energy states* that represent the available energy in the system. The Eon runtime system executes only those flows that the Eon programmer has marked as suitable for the

given energy state. Thus, an Eon programmer can easily write programs that provide different functionality or data quality based on current and future energy availability.

This flow and energy state information enables *automatic energy management*, allowing the runtime system to handle the complexities of adaptation. In response to changes in energy, the Eon runtime system dynamically adjusts the execution rate of flows and enables or disables application features. Because Eon programs describe energy abstractly (e.g., “high” and “low”), they are portable to hardware platforms with arbitrary energy profiles. The language itself is also highly portable: the current Eon compiler generates code for a variety of embedded platforms and operating systems, including Linux and TinyOS.

To demonstrate Eon’s utility and portability, we have built and deployed several Eon-based perpetual systems, including two solar-powered systems: one for tracking turtles and automobiles using GPS and another for capturing and transmitting images from remote locations. To quantify the ease of programming perpetual systems in Eon, we conducted a user study showing that programmers, who had just learned Eon, outperformed a control group using C—taking only 0.25% as much time to produce equally efficient code.

Outline: The remainder of this chapter is organized as follows. First, Section 4.1 describes the Eon language, focusing on the description of flows and energy states. Next, Section 4.2 describes Eon’s automatic energy management algorithms. Section 4.3 describes implementation details of the hardware and software systems, including the compiler, runtime system, and the trace-based simulator that the compiler can generate to predict performance before deployment. Section 4.4 describes three Eon-based perpetual systems we have built. Section 4.5 presents empirical results both for our user study and for one of the perpetual systems deployments. Finally, Section 4.6 discusses the most closely-related work.

4.1 The Eon Programming Language

Eon is a domain-specific language intended to support a broad range of perpetual systems. These include energy-limited systems that follow an event-response model of operation, such as devices that respond to external stimuli or to periodic, internally created interrupts. Eon combines both simplicity and elegance: its goals are to make energy-adaptive systems simple to write and easy to understand and to enable the use of optimized energy-aware runtime systems that automatically choose the best sustainable level of service.

An Eon programmer writes code that describes the sequence of operations that follows in response to external events and the desired adaptation policy, i.e., which sequences (flows) correspond to higher or lower power energy states. The Eon runtime system measures energy consumption and predicts the probable energy costs of each operation, the probable workload in the system, and the probable amount of energy the system will harvest. The runtime system then automatically adjusts the execution of flows for each energy state as indicated by the programmer.

It would be possible to build Eon's energy-aware features into either an entirely new general-purpose programming language or as extensions to an existing language. The first approach would require programmers to learn a new language while muddling basic constructs such as loops and conditionals with policy. This approach would also preclude the reuse of the vast amount of code already written in general purpose languages. Using language annotations, on the other hand, would simplify adoption for new programmers; however, the annotation syntax would have to be adapted to each new language. The resulting system would still mix the issues of adaptation with program logic. Most importantly, conventional programming languages do not explicitly manage program flows: these are implicit in program execution, and thus difficult to annotate.

Instead, we have designed Eon as a *coordination language* [74] that ties together code written in a conventional programming language, like Java, C, or nesC [29]. This approach provides programmers with a high level of abstraction that separates the details of energy

adaptation from program logic without sacrificing the reuse of existing code. Eon currently supports a range of different languages (C/nesC) and operating systems (Linux/TinyOS).

This approach also makes it simple to port an Eon program to a new platform. For example, porting an Eon program from an XScale-based device to a mote-class device required only modification of the platform-specific code used to implement the program logic. This portability makes Eon a natural candidate for use in embedded devices, given the wide variety of platforms, operating systems, and languages currently in use.

4.1.1 Basic Eon Syntax

A coordination language describes the flow of data through different components. We have built Eon on top of an existing coordination language called Flux [12], due to its features, simplicity and available compiler tools. Flux is a declarative language that describes a directed acyclic graph embodying the flow of data through the program. Flux *sources* connect to abstract nodes, which consist of a series of concrete nodes. Concrete nodes correspond to implementations written in a conventional programming language. Flux also allows for conditional flow through a program—a feature that Eon leverages for energy adaptation.

We illustrate Eon's syntax using examples from Figure 4.1 and the graphical representation of the program in Figure 4.2. We first describe the parts of the program that are the same as in Flux, and then describe Eon's extensions.

Flux-based syntax: As in Flux, an Eon programmer first declares each *source node* in the program and what types of data it outputs, such as `ListenBeacon` on Line 7, which produces an output of type `msg_t`.

Program execution always begins with event sources, which feed data into other *concrete nodes*, which correspond to functions implemented in conventional programming languages like C and nesC. Each concrete node takes a set of input arguments and produces an output set of arguments. For instance, *GetGPS* (declared on Line 12) takes no input and

```
// Predicate Types
// SYNTAX: typedef PRED_TYPE PRED_TEST
typedef gotfix TestGotFix;

// Source Node Declaration
// SYNTAX: NODENAME () => (OUTPUTS);
ListenBeacon() => (msg_t msg);
GPSTimer() => ();

// Concrete Node Declaration
// SYNTAX: NODEAME (INPUTS) => (OUTPUTS);
GetGPS() =>
   (GpsData_t data, bool valid);
LogGPSData(GpsData_t data bool valid)
        => ();
LogGPSTimeout(GpsData_t data bool valid)
        => ();
LogConnectionEvent(msg_t msg) => ();

// Regular Sources
// SYNTAX: source NODENAME => NODENAME;
source ListenBeacon => HandleBeacon;

// Timer Sources
// SYNTAX: source timer NODENAME => NODENAME;
// Eon Timer Source
source timer GPSTimer => GPSFlow;

// Eon States
// there is always an implicit BASE state
stateorder {HiPower};

// Abstract Nodes and Predicate Flows
// SYNTAX: ABSTRACT[[type,..][state]] =
// CONCRETE->...CONCRETE;
GPSFlow = GetGPS -> StoreGPSData;
StoreGPSData:[*,gotfix][*] = LogGPSData;
StoreGPSData:[*,*][*] = LogGPSTimeout;

// Abstract Node using Energy Predicates
HandleBeacon:[*,*][HiPower]
        = LogConnectionEvent;

// Eon Adjustable Timer
GPSTimer:[HiPower] = (1 hr,  10 hr);
GPSTimer:[*] = 10 hr;
```

Figure 4.1. A condensed version of Eon source code for the tracking application used in TurtleNet

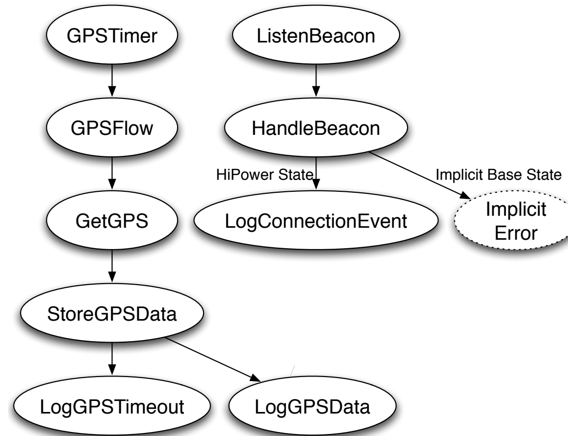


Figure 4.2. A graphical representation of the simplified turtle tracking code, shown in Figure 4.1

produces two output variables: a `GpsData_t` and a boolean. The Eon compiler checks to ensure that output and input types match in each flow.

Abstract nodes describe the flow of control and data through multiple concrete or other abstract nodes. For instance, GPSFlow (defined on Line 36) is an abstract node that is the combination of two other concrete nodes.

Conditional flows are implemented in Eon using *predicate types*: programmer-defined boolean functions that are applied to a node's output. In Figure 4.1, the *StoreGPSData* abstract node specifies two possible execution paths on Lines 37 and 38. By applying the *gotfix* predicate to the output of *StoreGPSData*, the Eon program decides which path to take. The test is defined on Line 3.

Multiple paths in an Eon program have semantics similar to those of `switch` statements in C. Paths are tested in the order that they are listed in the code, and the first matching path is chosen.

Each of the concrete nodes and all predicate tests must be implemented by the programmer in a supported conventional programming language (currently C or nesC). The

Eon compiler generates a set of stub functions for each node that must be implemented by the programmer.

4.1.2 Eon Extensions

While the parts of Eon drawn from Flux lets programmers define the sequence of operations that follow from events, they lack any method to express runtime adaptations. In this section, we describe how Eon extends Flux with constructs that describe what runtime adjustments to make as well as the priority with which they should be applied. The Eon application is then mapped to an adaptive runtime system, which continually adjusts the application in order to balance the demands of fidelity and sustainability. We continue to use the application shown in Figure 4.1 as an example.

4.1.2.1 Power states

Adaptation policies could be expressed as a set of *utility* functions describing the relative value of flows, and the rate of flows in an Eon program [13, 56]. Our own experience in building adaptive applications as well as anecdotal evidence suggest that general utility functions are difficult for programmers to use or understand.

In contrast to previous approaches, we have found that a simple partial ordering of flows and rates is sufficiently expressive. While a utility function can express a greater number of policies, such as non-monotonic functions, and are amenable to a great number of interesting analytical results, their usefulness is questionable while severely complicating life for the programmer.

In an Eon program, a programmer specifies an adaptation policy as a collection of behavior adjustments organized in a *state ordering*. An adjustment is declared simply by listing it in the state ordering, and its priority corresponds to the row in which it appears. All adjustments on a given row are applied together.

Figure 4.3 shows how the sample application's operating states are derived from the state ordering. An implicit BASE state (S_0) represents the program running without apply-

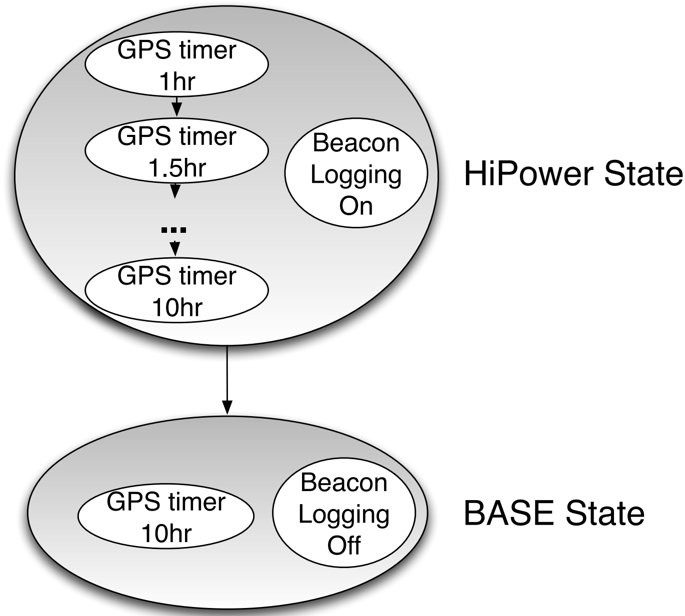


Figure 4.3. Sample State Order.

ing any adjustments. Subsequent states are defined recursively by applying an additional level of adjustments to the previous state (i.e., $S_i = S_{i-1} + L_{i-1}$). Also, a higher operating state is assumed to be more desirable and more energy-intensive than all lower states.

The state ordering of an Eon program defines which operating states can be chosen by the runtime system as well as their relative priorities. In addition to declaring adjustments, the system designer must also define what those adjustments are.

4.1.2.2 Adaptive Timers

One of the most common adjustments used to reduce energy consumption is to periodically turn off energy-hungry components, such as radios [4, 68]. In the turtle tracking application, the GPS receiver consumes two orders of magnitude more power than all other components combined. This cost makes the frequency of GPS readings the most important factor in the life of the device. Adaptively adjusting the duty cycle of a component or task represents a trade-off between application fidelity and energy consumption.

Duty-cycle adaptation is implemented in Eon using a special type of event source node called an *adaptive timer*. Adaptive timers differ from other sources in that they are not concrete nodes and are not implemented by the programmer. Instead, the programmer specifies a range of acceptable timer intervals. For example, the *GPSTimer* in the turtle application can fire anywhere from every hour to once every 12 hours. The interval is then set by the runtime system.

4.1.2.3 Energy-State Based Paths

Another common way to trade value for energy is to change the fidelity of data and the availability of services. Lowering the quality of images, audio, or video reduces the energy a device spends transmitting. Energy can be conserved further by making some services unavailable. For example, a remote camera may store images locally for later querying or only stream metadata, instead of streaming the full images [45].

Fidelity and availability adaptation is provided in Eon using *energy-state based paths*. This concept is akin to the predicate types used for conditional flows except that instead of choosing paths based on output types, paths are chosen based on the energy state set by the runtime system. In the case of our turtle application, *LogConnectionEvent* is called when *HandleBeacon* produces any type and is in a state labeled *HiPower*. If the node is low on energy, it may enter the implicit BASE state and cease logging beacons from other nodes to save energy. *HandleBeacon* does not take inputs of the BASE state type, so the flow ends in an implicit error that has no side-effects. In this example, Eon lets the programmer express preference for local operations over providing services to other nodes when energy is low.

4.1.2.4 Implementing Concrete Nodes

Implementing concrete nodes with nodes that block on I/O is straightforward, such as `read()` in a C/Linux system: the programmer merely adds procedures that run until the

I/O is finished and then return. If concurrency is a concern, Eon can use Flux's features for the automatic generation of multi-threaded code [12].

Implementing Eon nodes for the nesC/TinyOS environment is less straightforward due to the prevailing use of split-phase, event-based semantics. Instead of a single blocking function, a TinyOS concrete node is implemented as a simple nesC component that provides a single "call" command and an asynchronous "done" event that is signaled with the node's return values upon completion. This allows simple nodes that consist of a single function as well as more complex nodes that perform split-phase TinyOS operations.

4.1.2.5 Discussion

One feature that we considered but rejected during the development of Eon was to implement fine-tuned adjustments in node fidelity. For instance, like timers, we could have provided an explicit adjustment in the fidelity of a node that performs an operation such as video encoding. The runtime system would then have been able to adjust this knob to adapt the fidelity of video encoding in a large number of steps.

However, our experience with adaptive systems has been that only gross levels of adjustment are used. Video is either high-fidelity, low-fidelity, or perhaps a level in between. While Eon's timers are finely adjustable, the semantics of timers and their resulting energy cost are both simple to predict and effectively linear. For instance, firing a timer twice as often will use approximately twice as much energy per unit time.

The energy consumed by a video codec would likely have a non-linear relationship to its resolution. Tuning the fidelity would thus have a corresponding non-linear effect on nodes downstream that transmit the video. Recall that one of our goals is to provide a language that is conducive to well-performing runtime systems. Without an accurate prediction as to what effect an adaptation will have, it is more difficult to select the correct operating point. To find such non-linear, and often noisy, relationships takes a great number

of sample points, each of which may be consuming too much or too little energy while the system runs.

Further, there are an unlimited number of power management optimizations that can be made in sensor systems, from wireless duty-cycling, to link-layer power-control, and CPU frequency scaling. Our standpoint has been that anything that can be automatically inferred from the program itself in a general and reasonably efficient manner, should be. Along these lines, we have considered a great number of features to add to the language, but have generally favored simplicity over features instead of building a language that can express every possible energy-management scheme. For instance, instead of providing timers that synchronize to a common time reference for a Synchronized MAC (S-MAC) duty-cycling [85], we use the low-power listen mode present in many modern sensor radios.

4.2 The Eon Runtime System

By using the flow descriptions in an Eon program, on-line measurements of the per-task energy costs and workload, and predictions about the amount of incoming energy, Eon's runtime system adapts program execution according to the program's policies. This adaptation is completely automatic, and requires no offline profiling, and minimal online measurements.

4.2.1 Design Goals

Two goals inform the design of the Eon runtime system. First, it should support a broad array of low-power platforms, such as Motes [67] and Stargates [81], powered by solar energy. Because microcontroller platforms have relatively small memory sizes, the runtime system must be constrained to perform few measurements online.

Second, the runtime system should not require any explicit training, such as measuring the system under simulated load in a lab. Not only is this process painful for programmers, it is also inherently brittle. For example, training might require repeated measurement ev-

ery time the program is changed or deployed on a new platform with new peripherals and is dependent on having good models of the expected workload. As long as in-situ measurement is sufficiently accurate, and can be done with low-overhead, online measurement is greatly preferable.

4.2.2 Energy Adaptation Algorithm

The runtime system executes an adaptation algorithm that chooses the ideal power state for the system to use, based on its measurements of energy consumption and production. The adaptation algorithm strives to provide the highest fidelity to the application while avoiding two undesirable states: an empty battery and a full battery.

An empty battery prevents the application from executing even high priority flows. In most devices, it also imposes a period of dead time for the system, during which the battery must slowly charge up to a minimal level before the device can turn on again. When the battery is full, any additional environmental energy that the system harvests is wasted and cannot be stored for later use.

From Eon's perspective, any state of the battery between these two states is effectively equivalent: the goal of the system is to consume energy at a rate equal to the rate of energy production. The battery's role is to act as a buffer, riding out periods of low energy production and storing excess energy.

The runtime system periodically makes a decision about the ideal power state for the system by searching the possible adaptations, such as timer frequencies and power states. Eon favors smoothness of adaptation and searches for a single static policy that is likely to be sustainable for a long period of time (horizon).

Eon can make large adjustments using the energy-state based paths, and smaller adjustments using the adjustable timers. Eon initially assumes that the system runs at the highest energy state with the minimum frequency for all of the timers. It then computes the amount of consumed and produced energy over a short interval T_i. Taking into account the current

state of the battery, if this power state would empty the battery, the system lowers the energy state (for instance, Hi-Power to Lo-Power), and then repeats. Once it finds a state that is sustainable over the short interval T_i, it looks further into the future to see if the rate is truly sustainable, examining time horizons $2^n \cdot T_i$ for $n = \{1 \dots N\}$ time intervals.

Once the system finds a sustainable energy state, it performs a binary search on the timers using the same time horizons to discover the exact sustainable policy. This search strategy ensures that the policy is sustainable both over the short and long term, without requiring excessive compute time. More weight is given to the short term, as the runtime system periodically reexamines the policy to adapt to changing workloads and energy dynamics. The entire process runs in just 100 ms on a Mica2 mote for our full tracking program with 31 flows and a horizon of half a year.

4.2.2.1 Energy Attribution and Consumption

For adaptation, the system must have an accurate model of its energy consumption, including the energy cost and frequency of each independent execution path, or flow, through the program. Each time an Eon flow completes, the runtime system updates an exponentially weighted moving average (EWMA) of the flow's energy cost. The system also estimates the originating source's firing frequency and the probability of each branch taken by the flow. In the example in Figure 4.1, there are four possible paths through the program, each with a different energy cost and frequency.

Measuring per-path energy consumption requires careful accounting and hardware support. One option is to use a Fuel-Gauge IC, like those included in many modern laptop, mobile phone, and PDA batteries; two popular example include TI's bq27000 and Maxim's DS2770. These chips measure the capacity of the battery and charge/discharge rates, including corrections for temperature, battery-chemistry, and aging effects. A fuel-gauge chip provides an averaged, coarse-grained view of the energy remaining in the battery and

the current rate of charge or discharge. While necessary, this information is not sufficient to distinguish between energy consumption and charge, as both occur simultaneously.

The Eon runtime system requires both a fuel-gauge chip and fine-grain current measurement to attribute energy to individual program flows. In our hardware platform, we use an integrated current sensor, which separately measures the rate of consumption. This hardware is accurate to within 0.6mA, sensitive enough to measure differences in current consumption due to radio, flash, or peripheral use by individual flows on a variety of platforms.

The runtime system samples the current once every second, while simultaneously tracking the start and end times of each node in the program graph. Based on the percentage of time that nodes from a particular flow were running, the runtime system attributes energy to the flow. The rest of the energy is attributed to the runtime system and to the idle energy consumption of the platform.

Given the amount of energy consumed by the program and runtime system, Eon estimates the energy production rate. Adding the energy consumption for a period of time to the loss or gain in battery capacity yields the energy production over that same period.

4.2.2.2 Energy Source Model

In addition to knowing how much energy each path consumes, adaptation requires a model of how much energy the system is going to receive in the future. While Eon is not tied to any one energy production method, we concentrate on solar power, which is particularly challenging. The amount of available solar energy is highly variable. It is also unpredictable, since predicting sun intensity is, in essence, predicting the weather.

The model we use in our prototype is an adapted exponential weighted moving average (EWMA) based prediction algorithm from Kansal, et al. [43, 44]. This model essentially predicts that the energy production in the following days will be similar to recent days. Eon measures the energy production over a day, and assigns this value as $E(t)$. It then computes

the expected value of $E(t+1)$ as $\alpha E(t) + (1-\alpha)E(t-1)$. This model masks the diurnal cycles inherent to solar energy harvesting and is simple enough for use in small embedded devices.

4.3 Implementation and Deployment

This section describes the details of the Eon software and its hardware support. In addition, it describes the details of three Eon deployments: a turtle tracker, an automobile tracker, and a remote imaging system. The designs for the hardware, as well as a release of the application code, compiler, and runtime system, are all available from our website (`http://prisms.cs.umass.edu/~sorber/eon`).

4.3.1 Software

The software implementation of Eon includes a compiler and runtime system, as well as a generator for a trace-based simulator.

4.3.1.1 Compiler

The Eon compiler is a three-pass compiler implemented in Java, using the JLex Lexer and the CUP LALR parser generator. It is based on the original Flux compiler [12], extended with support for Eon's energy management features. The first two stages of the compiler build a graph representation of the program and then decorate each edge with input and output types. The third stage links this intermediate code with the Eon adaptive runtime system and user-supplied code that contains the program logic.

Eon can be ported to new languages and architectures with minimal effort. Our current implementation has been ported to two different environments: an Intel/CrossBow Stargate [81] XScale Linux system, using nodes written in C, and an Atmel microcontroller-based TinyOS system using nodes written in nesC [29]. In addition, we have ported Eon to a number of hardware platforms, including the Mica2Dot, Mica2, MicaZ motes [67], the

TelosB mote [66], and the Shockfish Tinynode [16]. Finally, support for the new *tosthreads* threading library for TinyOS has recently been added.

4.3.1.2 Runtime System

The Eon runtime system measures and adapts to energy usage and production. At the start and end of every flow, the code generated by the compiler invokes a set of functions that interface with the hardware, perform predictions, and calculate a running state. The result then informs the rest of the runtime system which state the system will operate in. The size of the TinyOS runtime is 4850 lines of code, occupying 18 kbytes of program ROM. While running, the runtime system uses 900 bytes of RAM for an empty program, plus approximately 30 bytes of RAM for each independent path in the program, depending on the size of the arguments passed between nodes.

4.3.1.3 Trace-Based Simulator

The Eon compiler optionally generates a trace-based simulator. By feeding an energy trace and traces for external inputs, an Eon programmer can test different energy predictors, workloads, programs, and adaptation policies. During deployment, an Eon node collects measurements of solar energy, consumed energy, battery state, estimated idle power draw, estimated per-path energy costs, path probabilities, and source frequencies. All of this information is then used as input to the simulator. Additionally, we have found that the information recorded by the runtime system is extremely useful as an energy profiling tool. Although not as accurate as an external measurement tool, it has been crucial in identifying energy bottlenecks in our systems.

4.3.2 Hardware

Eon's adaptation algorithms require hardware support. We have built a new charging and energy management board that controls the solar charging of lithium ion batteries, measures the capacity of the battery with a Maxim DS2770, and measures the current

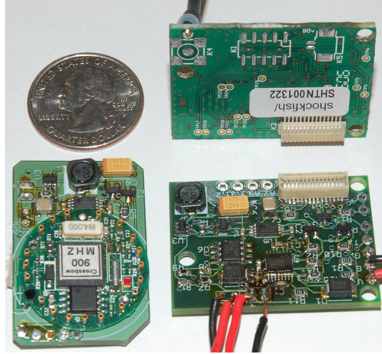


Figure 4.4. The two implementations of the energy measurement and charging board with a Mica2Dot and a TinyNode.

consumption using a Maxim DS2751. We have fabricated two versions of the board, one that accepts a Mica2DOT mote as a drop-in module, and one that attaches to a Shockfish Tinynode via a Molex connector. We adapted some parts of the hardware design from the Heliomote project [49]. The same board can be used with the Stargate, by attaching the board via a mote.

Figure 4.4 shows the deployment platforms for the Mica2DOT and TinyNode, shown with battery and GPS. This board can handle a wide variety of solar cells, ranging from a small, 25mA peak current cell up to a cell producing 2A. Additionally, Eon requires no program or runtime changes when changing the size or number of solar cells, since it only tracks the amount of energy production.

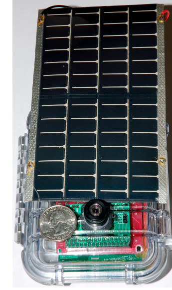
4.4 Deployment

In order to evaluate Eon, we have built several energy adaptive systems: a turtle tracking node, an automobile tracking node, and a remote imaging system. The evaluation section focuses on the automobile tracking system, and we describe all three systems here. We have also constructed a solar-powered WiFi web server on the Stargate platform.

While these deployments are somewhat limited in their scale and duration, we have gathered sufficient data to demonstrate Eon's utility in performing energy adaptation. Per-

(a) Early Eon node deployed on a Snapping Turtle.

(b) A more recent Eon node deployed on a Gopher Tortoise.



(c) Camera

Figure 4.5. Photos of two of the test applications, a turtle tracking device, and a remote camera.

haps more importantly, these deployments have driven the development of Eon, rather than following as a consequence of it. The applications have informed which features to add to the language, runtime system, and hardware support.

4.4.1 Turtle Tracking

As described previously in Section 3.1.1, much of the development of Eon is inspired by wildlife tracking and specifically, turtle tracking. In order to meet these challenges, we have designed and built turtle tracking devices and programs to run on both the CrossBow Mica2DOT and Shockfish TinyNode platforms. The turtle node includes a SiRF Star III-based GPS Receiver, an Ultralife UBC581730 250 mAhr battery, and one or two 5V solar cells. The Mica2DOT node, used in our first deployments, was packaged in shrink-wrap tubing with the ends sealed with a waterproof epoxy. The TinyNode-based device, designed for use on Gopher Tortoises, uses an epoxy-sealed custom plastic enclosure.

The design of the node is primarily driven by form-factor. The node must weigh less than 5% of the tortoise's body weight—limiting each device to roughly 50 grams—and fit without significantly increasing the height of the shell. Figure 4.5(b) shows the Eon node mounted on a gopher tortoise and Figure 4.5(a) shows an early version deployed on a Snapping Turtle.

Unfortunately, our initial deployment took place at the end of an unusually cool fall. The turtles prepared for hibernation early and spent a large amount of their time immobile and underwater, not emerging until the next summer. We thus collected relatively little data from the turtles: five days of solar traces and a handful of GPS locations.

Despite this small amount of data, we learned new facts about turtle behavior that were useful from a zoological perspective and that have led to improvements in our system. In particular, we discovered that the turtles were underwater 98.5% of the time. Because GPS does not work underwater, we added a water sensor to the node that lets the programmer specify that no GPS readings should take place if the turtle is underwater. In addition, we found that the turtles receive a great deal less energy while underwater, so little that even our upper-bound for the GPS timer was not sufficient to let the node survive. The combination of these two fixes should allow the node to survive long periods of time underwater.

4.4.2 Automobile Tracking

As a proxy for the turtles, we have also performed a second deployment using automobiles. We used the same hardware, adaptation policy, program, and runtime system, and collected two weeks of data from five devices mounted on the roofs of cars. The weather for that two weeks was highly variable, with several days of consecutive cloudy weather. These traces can be extended by looping them, which gives us a good idea of how Eon adapts to changing conditions. In addition, this automobile-based deployment has led to bug fixes and other improvements to the runtime system.

While we plan to redeploy the turtle nodes in a large-scale experiment in the summer, the evaluation we present here is based on data gathered from the automobile-based experiment. The complete application, excluding Eon runtime code, is 7900 lines of code. The complete system, including the Eon runtime, compiles to 42 Kbytes of program memory, and runs in 3600 bytes of RAM.

4.4.3 Remote Camera

Finally, we have built a remote camera application that demonstrates Eon's versatility. This application was inspired by various remote image applications at James Reserve [58], in SensEye [45], and at the Virginia Coast Reserve (VCR) [22]. Of note is the fact that VCR researchers programmed their cameras to scale their frame rates to cope with fluctuations of gathered solar power.

To ease their programming burden, we have constructed the video system using a TinyNode [16], a CMUCam low-power camera [70], a 400-mW-peak solar panels, and a 1 Ah battery, shown in Figure 4.5(c). The Eon application trades off the competing concerns of the frequency of image capture, and image streaming(high power), and image storage (lower power). Using the TinyNode's XE1205 radio, the images can be streamed from the solar-powered node to a base-station up to 1 kilometer away.

Once the CMUCam was connected, building a fully adaptive application took a single developer only three hours to build. No modifications were required to handle the larger solar cells or the energy requirements for the new platform and camera.

4.5 Evaluation

Our primary goal in designing Eon was to provide a simple language for building efficient energy-adaptive embedded systems. In this section, we evaluate the Eon language with respect to both usability and system performance.

4.5.1 User Study

To evaluate Eon's usability, we conducted a user study. Nine programmers were recruited for the study, the majority from a junior-level operating systems course and all having at least 4 years of prior programming experience. None had any prior knowledge of Eon and all were familiar with C. Each subject was initially asked to provide a self eval-

uation of past experience and programming expertise, which was used to divide them into two balanced groups, one using Eon and the other using C.

Each test subject then completed the user study individually. Participants were first given a 45-minute long tutorial covering the programming tools and computing environment, and an overview of energy-aware embedded systems. Following the tutorial, each subject was asked to write two applications.

The first application was a simple sensor application which periodically samples a sensor, stores the collected sensor readings, and answers simple network requests for past readings. The second application was an extension of the first application to make it adaptive, with the goal of providing the best sampling rate that their device could sustain without running out of energy. After completing these programming assignments, each participant was asked to take a post-experiment survey, qualitatively evaluating their experience.

The students performed the study in a simulated environment that included APIs for measuring energy spent using Flash memory, the radio, and for taking sensor readings. Also, in order to provide a fair comparison, we provided the C group with the same solar energy predictor used by the Eon runtime system. The build environment was instrumented in order to collect a snapshot of each participant's code with every successful compile. After the study was complete, we tested each submission. The initial program was tested for correctness to verify that it gathered sensor readings and answered queries correctly. The adaptive code was evaluated in terms of how well it was able to adapt to the provided solar trace.

Also, in order to remove transient behavior and focus on steady state performance, each submission was run for three months of simulated time, and results were only collected for the last month.

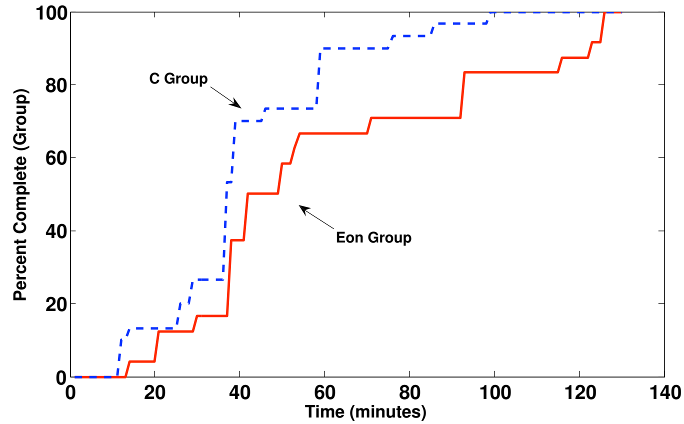
Figures 4.6(a) and 4.6(b) show the results of the user study for the first and second applications, respectively. In Figure 4.6(a), the progress of each group is shown with re-

spect to time spent (in minutes). Progress is measured as the percentage of correctness tests passed, with 100% meaning that all members of the group had passed all test cases.

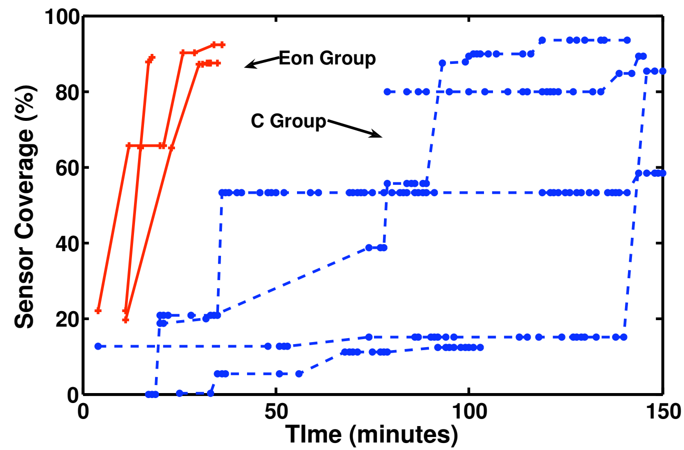
One striking feature is the similarity in progression between experienced C programmers and first-time Eon programmers. Members of the Eon group commented that the primary difficulties were learning Eon's syntax and understanding how Eon sources and flows are executed. Some commented that once these details were overcome, Eon provided a simple and intuitive programming style. We believe that the small difference between groups can be attributed to the Eon group's initial unfamiliarity with the language. We expect that experienced Eon programmers would be able to produce correct code for non-adaptive applications at least as quickly as programmers using a conventional general purpose language.

However, the results from the second application, shown in Figure 4.6(b), demonstrate the clear advantages of using Eon when building energy-aware software. This figure shows the performance of user submissions over time in terms of *percent coverage*. Every time a sensor reading is taken, an arbitrary amount of time t before and after the reading is considered to be "covered." The figure shows the percentage of time that was covered by at least one reading. For this experiment, we chose t such that the highest sustainable sampling rate would provide 100% coverage. Choosing a rate that is either too aggressive or too conservative results in reduced coverage. In this figure, we plot an individual line for each study participant. We also plot the *best solution so far* to make the figure easier to understand.

Unlike the results from the first stage, Figure 4.6(b) shows a substantial difference between the two groups. All members of the Eon group achieved 90% coverage within 40 minutes, while the C group lagged behind both in programming time and coverage. The Eon group's solutions were also uniformly good; this result stems from the effectiveness of Eon's runtime system. Three of the five members of the C group eventually achieved performance comparable to the Eon group, but took between 90 to 140 minutes to develop



(a) Experienced C programmers compared with first-time Eon users programming a simple sensor application. Aggregate group progress is shown over time. Despite language differences, both groups' progression is surprisingly similar. Small differences are likely due to the overhead of learning a new language.



(b) Percent sensor coverage (best so far) is shown comparing C and Eon programmers. By separating adaptation policy from execution, Eon users were able to build high-performing adaptive sensor applications significantly faster than those using C.

Figure 4.6. User study results

their solutions. Two of these solutions were inspired by TCP's exponential back-off. The other two C programmers' best submissions achieved 60% and 12% coverage, respectively. The longer programming times, high variance, and user comments all demonstrate the difficulty of writing adaptive software in conventional programming languages, even on a simplified sensor platform that avoids many common real-world complications.

4.5.2 Adaptation

One of the primary benefits of Eon is its ability to adapt the rate of flows in a program based on its currently available and predicted energy supply. Here we compare Eon's performance against several other possible systems and across individual devices.

To provide a fair and realistic comparison, we use trace-driven simulations based on data collected during the two-week automobile deployment. During this deployment, each of the five nodes collected hourly measurements that we then fed into our trace-based Eon simulator. To avoid measuring transient behavior based on the initial battery state and to show long-term behavior, we loop the measured traces to extend our simulations from two weeks to three months, and report only the results for the last month. Each test was run five times, and the simulator generates the amount of energy used by the GPS drawn from the distribution gathered from each trace.

In each test case, we change the GPS sampling rate according to five energy policies: a *conservative*, static policy based on the minimum sustainable rate across all of the traces; a similar, *greedy*, static policy based on the maximum sustainable rate; a *best sustainable* rate taken for each device individually; *Eon* using the solar predictor algorithm ($T = 24$); and *Eon (Oracle)* that uses a perfect weather predictor that knows the exact amount of solar energy that can be harvested in the future. Note that the conservative policy is an over-provisioning implementation that a system designer may try first: collect traces, find the one that gave the least amount of energy, derive a static policy, and use that policy on all of the devices.

The results presented in Figure 4.7 show the average rate of GPS readings. The error bars represent the standard deviation of the rate *within* each trace averaged over the five runs. This demonstrates the variability of the policy over the duration of each run.

The results in Figure 4.7 show that a conservative policy, unsurprisingly, only performs well for device 5, from which the policy was derived. The rest of the devices pay a large opportunity cost for not using a more aggressive policy. The Eon (oracle) policy, best sustainable, and Eon provide similar average results. However, Eon shows more variance, demonstrating that inaccuracies in prediction in energy harvesting leads to a larger range of rates. It is important to note that neither the best sustainable policy, nor the oracular system can be realized, as both require advance knowledge of future solar trends.

We initially found it surprising that the Eon predictor would do as well as the oracle. However, a closer look reveals that given the size of the battery in the system and the typical rate of consumption, a full battery will last for five days. This lifetime means that the solar power prediction does not need to be extremely accurate day-to-day, as long as it is accurate on average. In systems where the ratio of consumption to battery size is higher, prediction algorithms have more impact. Lastly, the greedy system exhibits a high average rate for most of the traces, but its variation is high. This variation is because the node often ran out of energy, dropping the rate to zero for long periods of time.

Figure 4.8 shows a more detailed view of the results from the same experiment. The stacked bars show the breakdown of how energy was spent by the different policies. The percentage numbers at the top of the bars show the average amount of time during the trace the device had a dead battery. The board overhead is the energy spent in the measurement board, the idle is the energy spent by the mote while not executing a flow (e.g. the overhead of the runtime system and the cost of an idling mote). The GPS energy was spent on taking samples, unused energy was energy left in the battery, and wasted is any energy that was collected, but could not be stored due to a full battery.

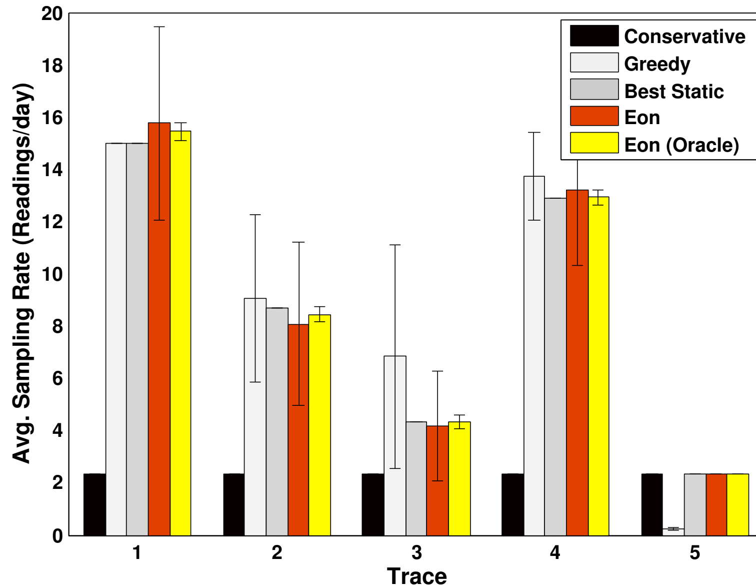


Figure 4.7. The average number of daily GPS readings taken are shown for different energy policies and energy traces. Despite large variations in energy supply, Eon is able to accurately approximate the best sustainable energy policy.

This graph shows the chief shortcoming of the greedy policy: aggressive use of energy leads to large periods of dead time. While sparse and bursty readings are generally undesirable, there is a more serious downside: the inability to run higher-priority flows. As we show in later experiments, when the program contains more than one flow, the dead time caused by one flow's overuse negates any prioritization the program may need.

4.5.3 Impact of Energy-State Based Paths

To examine the usefulness of energy-state based paths, we conducted a longer experiment using the remote camera application. Rather than conduct a year-long deployment, we used the solar cell to collect adequate solar traces for simulation, and then lengthened those traces using solar intensity data from the US Climate Reference Network, National Climate Data Center, and NOAA. By constructing a model that maps solar intensities to the power produced by the solar cells, we were able to extend the trace backwards for years' worth of data. Note that this process only works for the stationary camera. Long-term

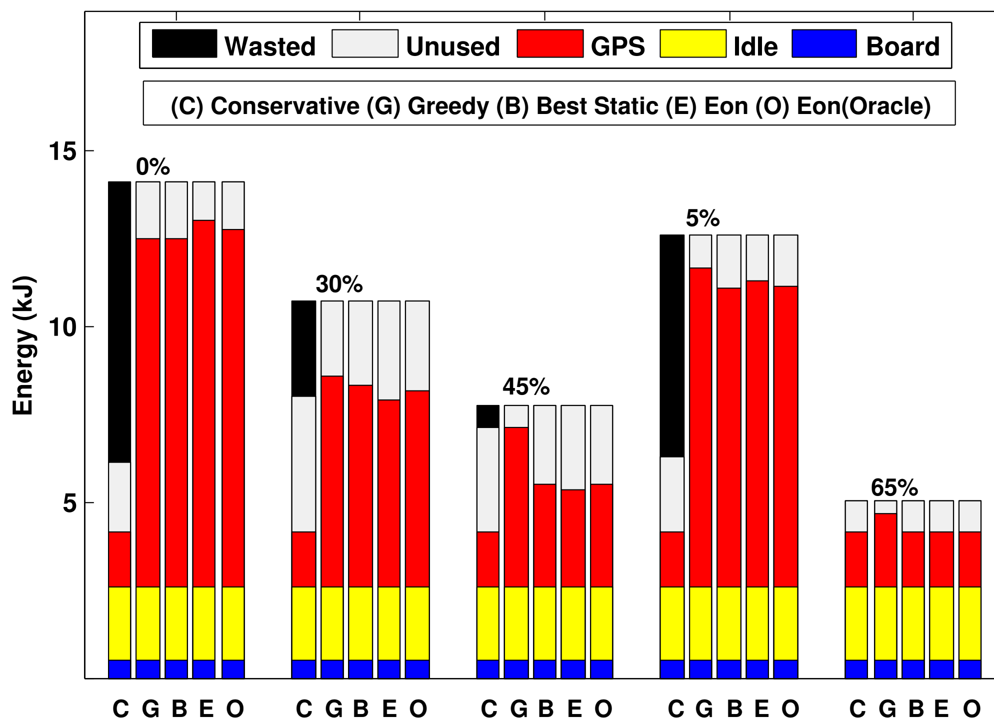


Figure 4.8. This figure shows the amount of each trace's energy that is consumed by different parts of the system. The percent dead time is also shown for traces that are not sustainable, above the corresponding bar.

simulation of mobile nodes requires information about each node's mobility as well as the weather, to accurately determine its energy budget.

Using the energy profiles collected from a running camera system and generated by our simulator, we compared the behavior of Eon against two systems, one that uses a fixed rate of 7 frames per hour (Fph) and one that uses 2.4 Fph. We then determined when each system had a dead battery, and thus could not respond to any queries for old images, and when the Eon system switched into its querying-only mode. The resulting frame rates, dead times, and querying-only times are shown in Figure 4.9. Note that the Eon system experienced no dead time for the entire trace, so we only plot dead time for the two static policies.

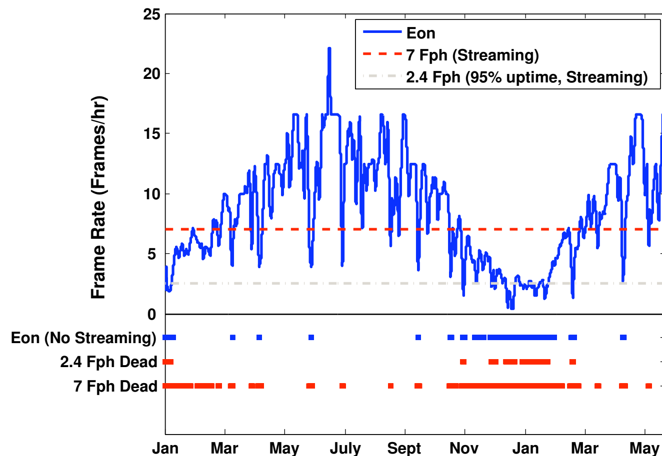


Figure 4.9. Frame rates for a remote camera application are shown over a 16-month trace, comparing Eon to two static policies. Periods of time when the Eon camera disabled streaming as well as periods of dead time for the static policies are shown across the bottom.

The results show that Eon can completely avoid dead battery times by adaptively switching into a query-only mode, while simultaneously lowering its frame rate. Note that it would be trivial to adjust this policy in Eon, e.g., to prefer higher frame rates over streaming. Without the ability to adapt, a fixed frame rate system may remain completely unavailable for months at a time. Eon is also able to scale its frame rate in tune with the seasons and short periods of cloudy weather.

4.5.4 System Overhead

In this section, we discuss the energy and compute-time overhead incurred by using Eon on our turtle/automobile monitoring node.

Since energy is the key focus of Eon, the energy overhead of the system must be kept to a minimum. Here, we measure the energy costs of several operations performed by the runtime system. We measure current draw using an Agilent 54621D oscilloscope, measur-

Operation	Overhead Costs	
	Energy	Time
Path Init	0.6μJ	0.3ms
Edge	1.4μJ	0.8ms
Path Cleanup	5.4μJ	2.1ms
GPS Reading	1–100J	20–400s
Evaluate State	0.5–2.0mJ	50–100ms

Table 4.1. Measurements of Eon overhead in comparison to GPS readings.

ing the voltage drop across a 1-Ohm sense resistor. We integrate the trace to determine the energy cost of the operation. Figure 5.4 presents these energy measurements.

Periodically reevaluating the energy state, which presents the largest single energy cost, varies widely depending on the structure of the application graph and the state of the system. If, for example, the battery is low and little energy is expected, the algorithm will quickly rule out higher power states. More complex applications will also take longer than simple applications since they have more flows to consider.

As Table 4.1 shows, the turtle tracking application requires up to 2.0mJ in the worst case to choose an energy state. However, since state evaluation happens only once per hour, this cost is easily amortized, resulting in an increase of only 2μW in the average power of the device. There is also a fixed overhead incurred every time a path is executed, which is equal to $(6.0 * 1.4N)\mu$ J where N is the number of edges in the given path. In comparison with the cost of taking a GPS reading, this overhead is insignificant, since it is at least 6 orders of magnitude smaller.

4.5.5 Measurement Accuracy

The runtime system's ability to accurately estimate the cost of individual paths in the program graph is vital to being able to make accurate adaptation decisions. We evaluate this accuracy by comparing measured task costs with the system's corresponding estimate

for tasks that consume different amounts of energy. In this experiment, we focus on small tasks (e.g., transmit data, write to Flash, etc.) that consume a few mJ and large tasks (e.g., GPS readings) that incur a high energy cost.

For small tasks (<100 mJ) the course grained averaging of our energy measurement board results in large errors in individual estimates (we observed up to 80%); however, averaging six consecutive estimates consistently yields an estimate within 10% of the measured value. For larger tasks (1–10 J) the Eon runtime system estimates the per-task energy cost to within 10% for individual task executions, and six consecutive estimates consistently results the measured cost to within 0.5%. The Eon runtime system benefits from this trend of increased accuracy for high-energy tasks. The penalty for mispredicting a small task is also small, and as these tasks are performed more frequently the system's cost estimate becomes more accurate. Mispredictions of large tasks, on the other hand, can have significant consequences on system lifetime, and cost estimates must be more accurate. Since large tasks are often performed infrequently, it is important to be able to provide accurate estimates with a small number of executions.

4.5.6 Impact of Battery Capacity

Our final experiment examines the impact that battery capacity has on Eon's ability to adapt, and on the cost of prediction errors. This experiment is set up as described in Section 4.5.2, except that we vary the size of the battery and the number of solar panels used.

The results of this experiment, shown in Figure 4.10, demonstrate how prediction errors are magnified as battery size decreases. While a 250mAhr battery is able to mitigate prediction errors, those prediction errors translate into large amounts of dead time when a 50mAhr battery is used. Applications that require a very small battery due to size and weight restrictions should use either more accurate or consistently conservative energy predictors.

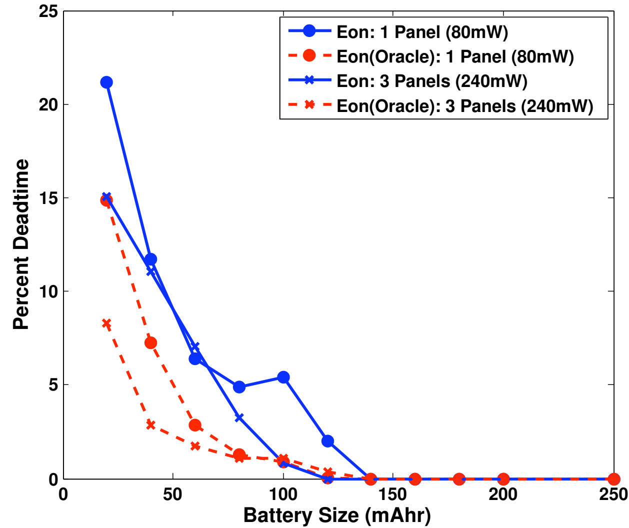


Figure 4.10. Device dead time is shown for different battery sizes for systems using one and three solar panels. Performance using Eon's EWMA predictor is compared with perfect energy prediction (Oracle). The benefit of better energy prediction is most notable when using a very small battery and the cost of prediction errors is greatest.

We note that an additional benefit of Eon's automatically generated simulators is the ability to use them to determine what size battery or solar panel to choose for a given deployment.

4.6 Related Work

Eon derives from a large body of work on energy adaptation in operating systems, as well as dataflow and coordination languages.

Languages/Programming Abstractions: To our knowledge, Eon is the first system that specifically targets energy adaptation at the programming language level. Eon's energy adaptation features are built on a dataflow-based, coordination language [31, 60]. Eon uses this dataflow abstraction to expose just enough structure to make building an adaptive runtime system possible. However, in contrast to many dataflow languages, Eon's goal is to simplify writing energy-adaptive programs, rather than expressing concurrency or real-time ordering constraints [7, 35].

Other languages and programming abstractions have been proposed in order to simplify the programming of embedded sensors. SNACK [32] provides language constructs that combine components written in NesC [29], in order to simplify and encourage code reuse. The Flask language [57] has been developed concurrently with Eon [74], and both languages share many properties: both are coordination languages that combine nesC modules together in an acyclic graph. However, unlike Eon, Flask does not provide support for energy adaptation.

Another difference between Flask and Eon is that Flask is a macroprogramming system, while Eon programs run on a single node. Macroprogramming languages and abstractions, like Flask, Bundles [79], Kairos [34], and TML [64] provide a more centralized approach to sensor network programming; however, these systems have ignored the challenges of perpetual systems.

Energy Application Adaptation: There has been a wealth of research on building systems that adapt to current conditions, including energy. Odyssey provided the seminal work in application-aware adaptation [65], and later work extended it to account for energy [27]. The ECOSystem project uses application adaptation to share energy fairly between applications, and governs that system's consumption rate [87]. In each case, energy-aware adaptation trades fidelity for energy savings to target a particular device lifetime. Eon builds on this concept by targeting perpetual operation, while expressing the adaptation policy as part of the program. This provides a much tighter integration of resources, programming language, and runtime system.

Since, Eon's publication a number of other adaptive sensing systems have also been built, which share many of its characteristics. Levels [47] was developed concurrently with Eon and also uses the idea of defining abstract energy states (e.g., Levels) that are tied to program adjustments. Instead of using a coordination language, like Eon, Levels provides a set of annotations that can be applied to existing NesC code. Levels also ignores energy harvesting, focusing on achieving predictable lifetimes rather than perpetual operation.

Another closely related system is Pixie [53], a sensor network operating/programming system that, like Eon, uses a data-flow programming model. Pixie uses a currency-based model (e.g., tickets), like ECOSystem [87], where tickets corresponding to resources (e.g., bandwidth, energy, etc.) are produced by resource allocators and consumed by application tasks. Managing tickets manually can be cumbersome, so Pixie also provides resource brokers that coordinate resource demands with available resources. Pixie and Eon have many differences and similarities; however, the most fundamental difference is the user's role. Pixie focuses on making resources visible and maximizing programmer flexibility, while Eon's goal is to make energy management invisible and fully automatic. Pixie programmers can create elaborate energy management policies, while Eon programmers ignore the energy management policy and focus on their application.

4.7 Discussion

In this chapter we have described Eon, a new language and runtime system for building self-adapting perpetual systems. Designed to be both expressive and simple, Eon eases the burden of building energy-adaptive applications. The Eon runtime system effectively manages changing energy availability and demands, while hiding most of the system's complexity.

Eon solves some of the challenges that are fundamental to perpetual systems; however, the system, as described, falls short in several ways. Specifically, Eon's focus is centered on balancing energy harvesting and consumption for a single device. In a network of perpetual devices, effective energy management must also account for interactions between devices. In the next chapter, we describe how Eon can be extended to address the needs of networks of perpetual mobile devices.

CHAPTER 5

TULA: FAIR AND BALANCED DATA DISSEMINATION

By including explicit control flow and energy policy information, Eon greatly simplifies the programming and deployment of individual energy-adaptive perpetual devices; however, perpetual systems are rarely useful when operating alone. Most applications that benefit from perpetual operation [20, 36, 37, 74, 88]—including mobile sensing and tracking—also require communication to make collected data available to scientists and other users for timely use.

Communication in networks of perpetual devices compounds the problems caused by uncertain energy harvesting and consumption. Perpetual mobile networks are often deployed without any prior knowledge of network topology or bandwidth constraints. Over time, changes in social networks and other external factors cause these networks to evolve, which can drastically affect both system performance and energy availability. As the network evolves, some nodes may never have a direct wireless link to the Internet, and must instead route data over uncertain mobile-to-mobile connections using disruption-tolerant networking (DTN) techniques [5, 39, 75].

Early perpetual systems, like ZebraNet [88], focused on minimizing energy consumption rather than energy awareness and adaptation. More recent systems either use local energy adaptation techniques without considering data delivery [47, 53, 74] or use adaptation techniques for purely static networks [24]. However, adapting to both energy and network variations is considerably more difficult. In particular, a node needs to adapt and balance both its sensing and routing tasks. In a long-running system the goal is to gather as much data from nodes as the limited resources of network bandwidth and energy permit.

Sensing more data than can be delivered by the network is not useful, while gathering less underutilizes the system's potential. Similarly, systems that depend on cooperative replicating routing protocols [5, 75] must balance the energy devoted to sensing and routing their own data with energy used to route data for other nodes.

In this chapter, we describe a system, Tula, that addresses this challenge for perpetual mobile sensor networks. A Tula node uses a distributed algorithm to balance energy allocation across three tasks: sensing, routing the node's own data and the routing data for other nodes. The Tula energy allocation ensures max-min fairness, which allows data collection from all nodes including poorly connected nodes. The key insight behind Tula is that sensing and routing are inherently dependent, and optimizing only one or the other in an energy-constrained environment is futile.

Given the allocation for sensing and routing, Tula uses an adaptive sensing system to collect data and a DTN routing algorithm to deliver the data. We formulate the Tula allocation problem as a constraint optimization problem (COP). Each Tula node measures energy consumption for sensing and communication and gathers data about the environment through node meetings, to locally solve the COP on an embedded device. Tula is general, and automatically adapts across mobility patterns, from static to highly mobile environments.

We evaluate Tula in the context of two example systems—TurtleNet and DieselNet—that we discussed in Chapter 3. We use traces of mobility and energy harvesting from TurtleNet and DieselNet, combined with an implementation of Tula on ShockFish TinyNodes [16].

Our evaluations over both TurtleNet and DieselNet show that Tula senses and delivers data within 75% of an optimal, oracular system that perfectly replicates data and has foreknowledge of future energy harvesting. The protocol is fair in terms of delivery rates across nodes, and it comes within 95% of the optimal in terms of the max-min fairness objective. Tula not only works well for sparse mobile networks, but also for static mesh networks.

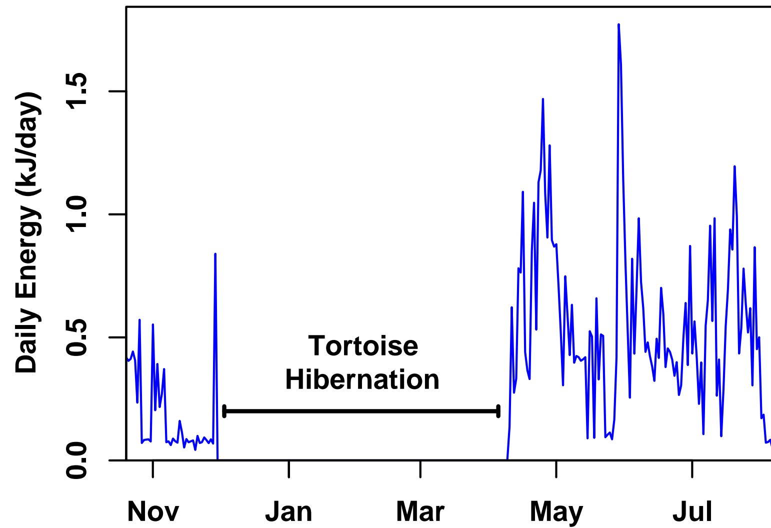


Figure 5.1. Daily solar energy is shown for a TurtleNet node before and after hibernation.

Our evaluations on a synthetically generated mesh network shows that Tula adapts well to the static environment and senses and delivers data within 85% of the optimal rates. Finally, we show that Tula can be implemented on a small microcontroller with modest code, memory, and processing requirements.

5.1 Challenges for Perpetual Networks

Devices that operate perpetually using harvested energy represent a new class of mobile system that promises to enable a wide and largely unexplored range of potential applications. This vision includes significant advances for scientists studying mobility in nature. In spite of decades-worth of study, the movements and behaviors of most animal species in the wild are completely unknown. Current methods like trapping and manual radio telemetry are labor intensive, yield few data points, and significantly increase the frequency of animal interactions with humans. By using small in situ sensor devices to observe animal location, movement, and environmental conditions, researchers will be able to collect more data at higher temporal densities with minimal impact on behavior. This shift promises to answer

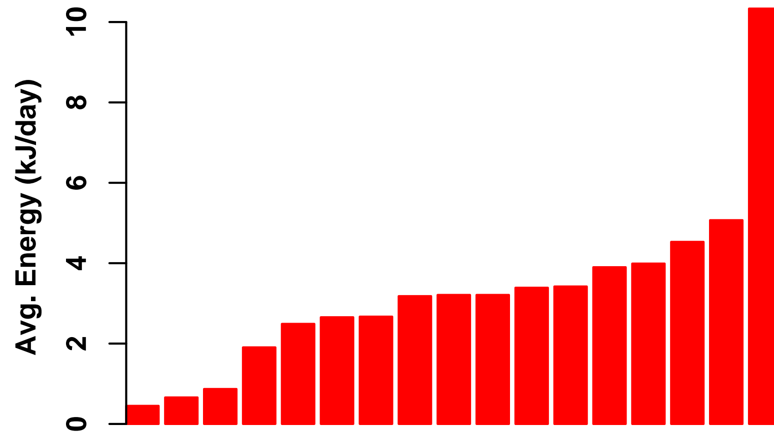


Figure 5.2. The average daily energy harvested by each TurtleNet node during a 1-month trace.

long-debated questions about habitat usage, population trends, and complex interactions between different species, including humans.

To explore these potential benefits and inherent challenges of perpetual networks, we have deployed TurtleNet as described in Section 3.1.1. Our deployment consists of 17 tracking devices attached to Gopher Tortoises (*Gopherus polyphemus*), shown in Figure 3.2. Each device consists of a Shockfish TinyNode, a solar panel, a battery, multiple sensors, and additional energy measurement hardware.

During the deployment, the devices record wireless connection opportunities with neighboring nodes and periodic sensor readings, including temperature, GPS coordinates, battery level, solar energy harvested, and energy consumption. Unlike traditional networks, these nodes rarely have an end-to-end connection to one of the two deployed GPRS-enabled base stations, and devices must opportunistically deliver collected data using mobile-to-mobile routing [5, 75]. When two mobile nodes are within communication range, called a *connection opportunity*, they exchange data. This data is stored and then forwarded during

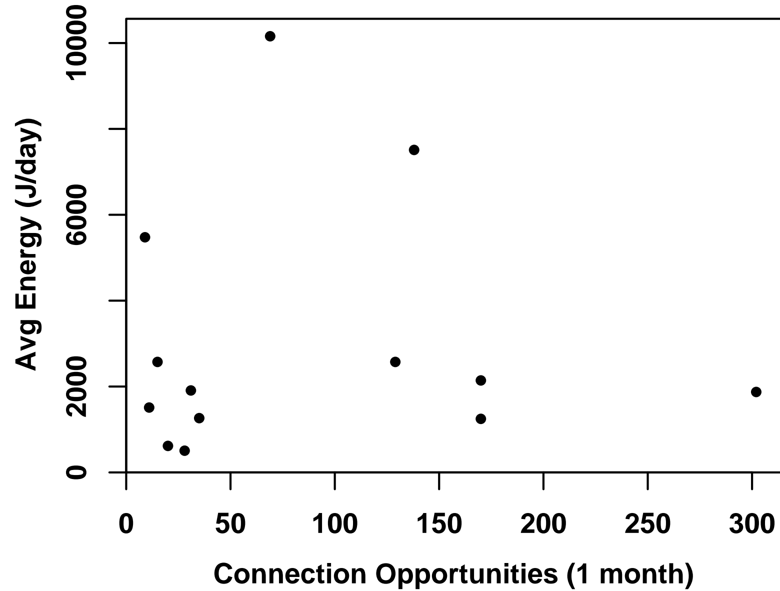


Figure 5.3. Harvested energy plotted against number of meetings for each node. Energy-rich nodes are not necessarily better connected and vice versa.

subsequent connection opportunities until it is eventually delivered to the sink. The network has been in operation since August 2008.

5.1.1 Challenges

On analyzing the data traces from our deployment, we uncovered a number of key challenges. The primary difficulty in designing TurtleNet—and generally any untethered mobile network—is the continuous variation of both energy harvesting and network connectivity due to mobility. Figure 5.1 shows how a node's daily harvested energy varied—experiencing both day-to-day and seasonal changes. Note that within a 10-day period, daily harvesting ranged from less than 0.1kJ to more than 1.7kJ. *In order to support perpetual operation, a device must adapt its behavior over time.*

In addition to temporal variation, energy harvesting also varies considerably across the network. Figure 5.2 shows the average daily energy harvested by all nodes in the network over a 1-month period of time, sorted to show the energy distribution. The figure shows that there is significant variation in energy harvesting across nodes. *With diverse energy*

budgets in the network, each node needs to balance its available energy between sensing and delivering data.

Recall that nodes rely on other well-connected nodes to store and forward their data to the destination. Unfortunately, there is very little correlation between how connected a node is and the energy it gathers, as shown in Figure 5.3. The well-connected hubs that are best positioned to route data may not have sufficient energy to support network demand. *Routing decisions in perpetual networks must depend on not only topology, but also the available energy.*

Finally, in TurtleNet—and most other mobile systems—connections between mobile nodes often exhibit patterns due to social habitats as shown in Figure 5.4. The figure shows that 40% of node meetings repeat more than 10 times a month. The meeting patterns are not completely random and can be leveraged to combat the network's uncertainty. In other words, if two peers have a connection opportunity, we can expect that the peers will have future connection opportunities.

5.1.2 Design Goals

In this chapter, we describe Tula, a system that addresses the above challenges by supporting perpetual operation and ensuring fair and efficient data collection. The goal of perpetual operation entails that energy spent sensing, storing, processing, and communicating must be matched with harvested energy. In addition, data sensing rates must be matched with that of delivery. For example, a node should not sense more data than can be delivered. In addition, Tula must operate in environments with sparse network connectivity and on platforms that are limited in energy and computational resources.

Finally, fairness is a critical function of Tula. Tula operates in networks with significant variation in available energy and network connectivity between nodes. Maximizing network throughput or minimizing delay, without enforcing fairness, will likely result in well connected and energy-rich nodes collecting and delivering their own data at a much higher

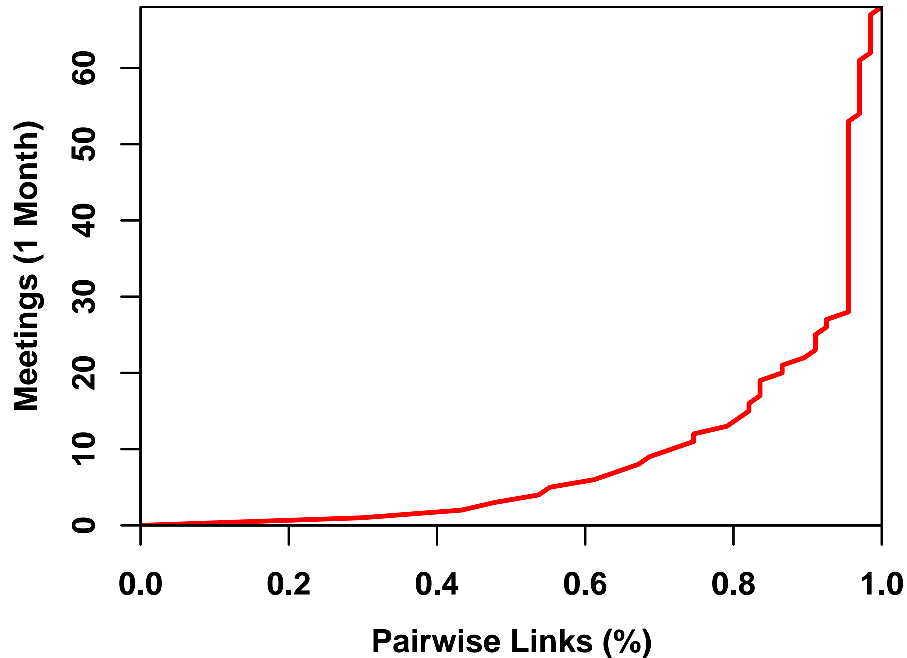


Figure 5.4. CDF of the pair-wise meeting frequency during 1 month of TurtleNet operation. While some meetings occur too infrequently to be very useful, 50% of the node pairs repeat at least 5 times.

rate, while starving nodes that are further away. Well-connected nodes may also have their energy budgets depleted by high network demand.

Different models have been proposed for sharing resources among nodes and flows within a network. For Tula we have adopted the goal of max-min fairness, which requires that a nodes' sending rate be improved only after all lower rates have already been maximized. This model is one of the most well-known network fairness models, and it fits well with wildlife tracking applications, including TurtleNet. Without fairness, perpetual operation is still possible, but a poorly connected node may deliver very little data. Wildlife tracking applications typically seek to characterize animal behavior and their interactions with the environment by collecting as much data from as many nodes as possible. This emphasis on sensing "breadth" rather than "depth" can be expressed using max-min fairness. Adapting Tula to other fairness models is left for future work.

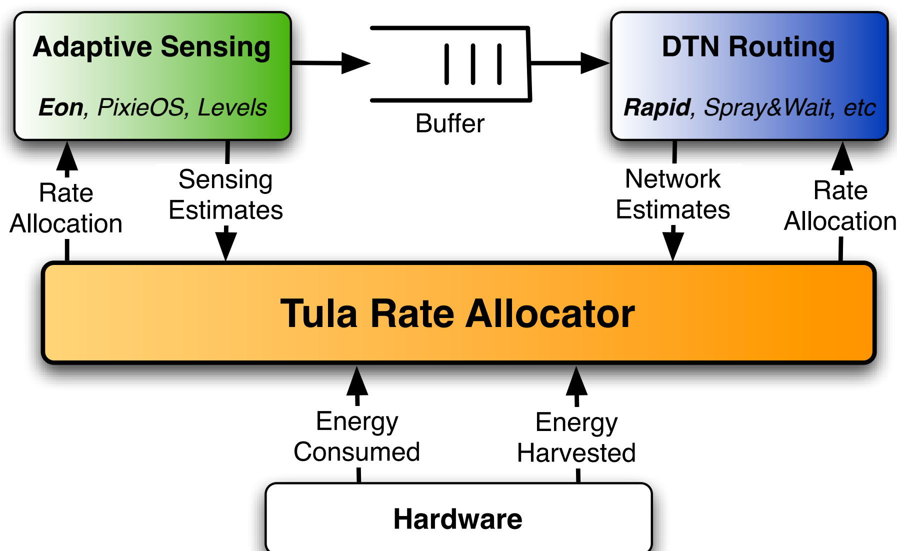


Figure 5.5. The Tula architecture.

5.2 Tula Architecture

The Tula architecture, shown in Figure 5.5, consists of three main components: an *Adaptive Sensing System* for collecting sensor data, a *DTN routing algorithm* for opportunistically delivering that data, and a *Rate Allocator* that coordinates both sensing and routing activities by appropriately allocating resources.

Adaptive sensing systems adjust application sensing rates alone in response to changes in a device's energy budget. Existing systems, including Eon [74], PixieOS [53], and Levels [47], estimate or measure the energy costs of various application tasks and automatically adjust application behavior to match a device's changing energy budget.

In sparse networks, DTN routing systems opportunistically route network packets from source to destination using sporadic and uncertain device-to-device meetings. Many systems have been designed, including Rapid [5] and Spray and Wait [75], to effectively deliver data over intermittent links while responding to changing network conditions.

Unfortunately, these systems are not designed to work together. Existing adaptive sensing systems consider only local energy constraints, ignoring the impact of sensed data on

the network. Similarly, DTN routing systems assume unlimited energy and consider only bandwidth restrictions. Tula's core function is to overcome this challenge by combining the benefits of adaptive sensing and DTN routing into a single coordinated system.

Rather than build a complete system from scratch, Tula abstracts the sensing and the routing systems and controls these systems using an allocator that balances energy for sensing with that of data delivery. The Tula energy allocation is most easily understood in terms of rate: the number of packets, or bytes, that can be generated by sensing and delivered by routing, over some time period.

The allocator's objective is to maximize the rate at which sensor data is collected and delivered, while ensuring that the allocated rates are fair to all nodes. To this end, Tula must appropriately adjust *(i)* the rate of sensing, *(ii)* the routing rate for the node's own data, and *(iii)* the maximum routing rate for each neighbor's data. Given the sensing rate, Tula leverages existing sensing mechanisms that adapt the local sensing task according to available energy. Given a routing rate, Tula adapts existing DTN routing protocols to route data within the given rate.

5.2.1 Adapting Sensing

Adaptive sensing systems change their sensing rates according to energy conditions. Adapting the sensing rate is especially important for perpetual operation in sensor nodes that have multiple sensors or have high variability in harvest energy. Eon [74], an adaptive sensing system that we use in Tula, uses hardware support to measure energy consumption and harvesting. Eon then combines the measurements with runtime information about the application to estimate the energy cost of various program tasks. Finally, Eon uses this information to determine how much energy is required to sense data at a given rate. This relationship is communicated to the rate allocator, which uses the information to solve the allocation problem (Section 5.3).

5.2.2 Adapting Routing

The Tula rate allocator only assigns the maximum rate at which a node can route data for each of its neighbors. The actual routing decision involves other tasks including estimating routes, tracking acknowledgements, and adapting the route to changing network connectivity. In Tula, we use the Rapid [5] DTN routing protocol and adapt it to an energy constrained environment.

Rapid estimates a distance metric between each node and the sink, where the distance is the expected delivery delay. Rapid then replicates data through multiple routes based on a marginal utility heuristic. Rapid estimates network parameters including the expected delay and bandwidth by averaging over a sliding window; it then communicates the estimates to the allocator.

The routing rate assigned by the allocator is only an upper bound. The actual data that is routed through the path depends on the quality of the route. For example, let nodes A and B be peers and let A's allocator assign a maximum rate at which it can route data for B. Because of changes in the network (due to mobility, interference, etc.), B may send data at a much lower rate through A; in turn A will reduce its rate allocation for B and balance the sensing rates appropriately. In other words, the routing protocol adapts to changing network conditions, that in turn affect the rate allocation. We discuss this mechanism in more detail in Section 5.4.1.

5.3 Rate Allocation

We first describe the rate allocation algorithm by making two simplifying assumptions: *(i)* A node routes packets through only one neighbor, and *(ii)* data is only forwarded, never replicated. Later, we describe how these assumptions can be relaxed. The network, shown in Figure 5.6, illustrates this simplified scenario. The allocation algorithm is described with respect to the node, n, with upstream nodes $u_1, \dots u_k$ routing data through n, and a single downstream neighbor d, through which data is routed toward a sink or base station.

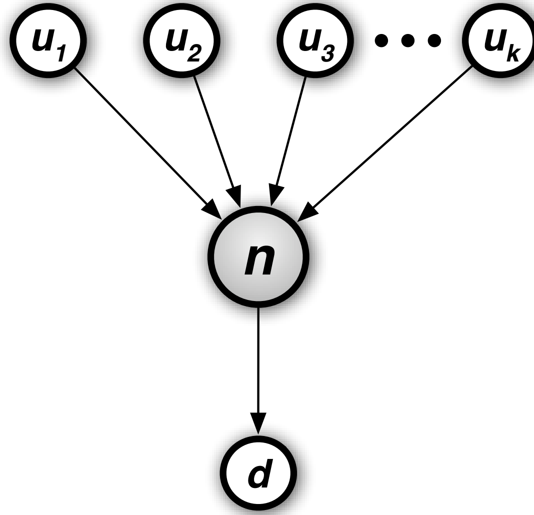


Figure 5.6. A simplified example to illustrate the Tula distributed allocation algorithm. The algorithm is executed by node n, whose upstream neighbors are $u_1, u_2, \ldots u_k$

Each node determines its set of upstream and downstream neighbors based on the routing protocol's *distance* estimate. In the Rapid routing protocol that we use, the distance is the expected delivery delay. When two peers meet, the peer with a lower delivery delay is the downstream neighbor, and the one with the higher delivery delay is the upstream neighbor. Section 5.4.4, discusses the implications of relaxing the definition of upstream and downstream neighbors.

Node n executes the Tula allocation algorithm in order to determine its own sensing rate r_n and the rate at which it can route data for its neighbors, $r_1, r_2 \ldots r_k$. The allocation problem is formulated as a *Constraint Optimization Problem (COP)* with the objective of finding a max-min fair rate allocation, based on a set of input variables that are either estimated locally or exchanged between neighbors during contact opportunities.

Locally measured values, shown in Table 5.2, include the energy required to collect a packet of sensor data (E_s), to receive (E_r) a data packet, and to deliver (E_d) that packet to the sink. P is the total energy harvested by the node. All of these measurements can be

obtained by a node's hardware while the device is deployed. In Tula, we use the low-level energy profiling capabilities already provided by the Eon [74] runtime system.

The network variables, shown in Table 5.1, are exchanged either *upstream* or *downstream* through the network whenever nodes meet. The direction for each variable with respect to n is shown in the table. Network variables are only exchanged with immediate peers and are not flooded across the network. These variables are described along with the COP formulation in the following paragraphs.

5.3.1 Objective function

The objective of the rate allocation is to achieve max-min fairness in the data collected across the nodes. A rate allocation is max-min fair if increasing any rate, r_i, requires the reduction of a lesser rate, r_j, $r_j \leq r_i$. This is also referred to as a lexicographically maximized rate assignment [24] (Equation 5.1).

$$\textit{Objective: Lexicographically maximized } \{r_1, r_2, \ldots, r_n)\} \quad (5.1)$$

5.3.2 Energy conservation constraint

Perpetual operation requires that a node's harvested energy be sufficient for all sensing and networking tasks. Equation 5.2 ensures that n can sense and deliver its own data at a rate of r_n and receive and deliver data at a rate of r_i from each upstream neighbor u_i without exceeding its power budget, P. All variables are estimated locally using hardware instrumentation.

$$\sum_{i=1}^{k} r_i(E_r + E_d) + r_n(E_s + E_d) < P \quad (5.2)$$

5.3.3 Downstream constraint

The total data that n can route is capped by its downstream neighbor d. In the same way that n assigns a maximum routing rate to its upstream nodes, d likewise assigns a maximum data rate to n. Equation 5.3 ensures that n will never accept or collect (by sensing) data at a rate higher than the rate, O, at which it can deliver data. Node n receives the value for O from d each time they meet.

$$r_n + \sum_{i=1}^{k} r_i < O \tag{5.3}$$

5.3.4 Upstream constraint

The objective function and the first two constraints alone will result in all upstream routing rates being assigned equal to the local sensing rate. This equal division of resources is fair; however, the system will be underutilized if some upstream neighbors are unable—due to energy or bandwidth limitations—to take advantage of the allocated rate. To avoid this condition, each upstream node, u_i provides node n with an additional value: B_i is the maximum amount of data that an upstream node can send, given its energy limitations. An upstream node u_i can compute its value of B_i by solving the COP without the downstream constraint (Equation 5.3)

$$r_i \leq B_i \, (\forall \, i \in [1, k]) \tag{5.4}$$

The COP can be solved using a well-known progressive filling algorithm [8]. The algorithm evenly adds rate to each upstream link. As rates reach their limits, they are excluded from receiving additional rate, and the process continues until either all peers are excluded or no residual energy is available. This algorithm is fast, easy to implement, and amenable to use on low-power platforms, as we show in Section 5.5.

F_i *(down)*	The fraction of node u_i's data that are sent through n
B_i *(down)*	The maximum rate at which u_i can forward data to n
O_j *(up)*	The rate at which n can route packets through its downstream neighbor, j

Table 5.1. List of inputs that are exchanged between n and its neighbors to solve the COP. Variables marked *(up)* are exchanged from n's upstream neighbors, and variables marked *(down)* are exchanged with n's downstream neighbors.

E_s	Energy required to sense a packets worth of data
E_d	Energy to deliver a packet
E_r	Energy required to receive a packet
P	Power available for sensing and routing

Table 5.2. Variables that are estimated locally by n to solve the COP

5.4 Incorporating Routing

The simplified Tula rate allocation makes assumptions that do not hold in practice, especially when using DTN routing to navigate a sparsely connected network. This section describes the effect of removing these assumptions, resulting in the modified COP, shown in Figure 5.7.

5.4.1 Routing through multiple nodes

DTN routing algorithms often rely on multiple downstream nodes to route packets. When presented with multiple downstream options, as shown in Figure 5.8(a), the routing algorithm on n determines which packets will be routed through d_1 and which will go through d_2. These routing decisions are limited, however, by the routing rates allocated by the downstream nodes. Therefore, the total data that n can route is now the sum of the

Objective

Lexicographically max. $$\left\{\frac{r_1}{F_1}, \frac{r_2}{F_2}, \ldots, \frac{r_{n-1}}{F_{n-1}}, r_n\right\} \quad (5.5)$$

Constraints

Energy conservation

$$\sum_{i=1}^{k} r_i(E_r + E_d) + r_n(E_s + E_d) < P \quad (5.6)$$

Downstream

$$r_n + \sum_{i=1}^{k} r_i \leq \sum_{j=1}^{m} O_j \quad (5.7)$$

Upstream

$$r_i \leq B_i \, (\forall i \in [1, k]) \quad (5.8)$$

Figure 5.7. Energy allocation problem formulation solved by node n. The goal is to estimate r_n, the local sensing rate and r_i, the rate at which n can route packets for each of its neighbors u_i

rates allocated by each downstream neighbor, $d_1, d_2 \ldots d_m$. We account for this limit in the COP by replacing the maximum downstream rate, O, with the maximum downstream rates allocated by m downstream neighbors, $O_1, O_2 \ldots O_m$. The new downstream constraint incorporating these variables is shown in Equation 5.7.

This change also impacts downstream nodes, illustrated in Figure 5.8(b), where n receives only a fraction, f, of the packets routed by u_1. The remaining fraction, $1 - f$, of the packets are routed through another node n'. Using the original COP, both n and n' would allocate resources to u_1 as though each were routing all of its data—clearly defeating Tula's efforts at fairness. Node n avoids this by allocating rate to u_1, proportionally to f. To accomplish this, we introduce a variable, F_i, which represents the fraction of all data routed by a node u_i through n. Equation 5.5 shows the modified objective using F_i to al-

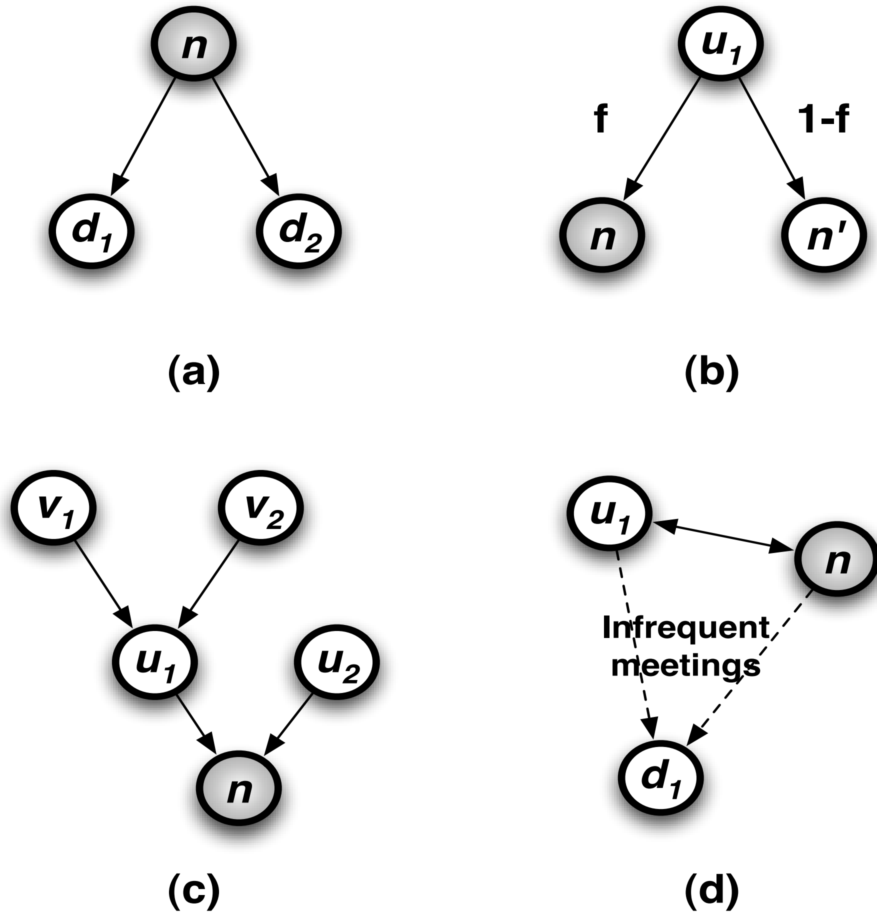


Figure 5.8. Scenarios that complicate the simple Tula allocation algorithm

locate rates fairly to fractional network flows. A node receives F_i values from its upstream neighbors, which keep track of these values by maintaining a limited routing history.

In addition to ensuring fairness, the F_i values also provide a mechanism by which an upstream node's routing protocol can express demand to a downstream node's rate allocator. For example, if the routing protocol on u_1 diverts packets from a less promising n to a more promising n', the fraction of data routed by u_1 through n will decrease, signaling the rate allocator at n to reduce its allocation. Alternatively, if u_1 wants to route more packets through n, it will communicate an increased F_1 value to n. This signals n to increase its allocation, so long as the increase does not violate the fairness model.

Another result of allowing multiple downstream routes is that allocation decisions at node, n, can indirectly effect nodes that are not on n's upstream or downstream path. Consequently, solving the optimization problem optimally would, in the worst case, require each node to have complete knowledge of the network. Obtaining this global knowledge is costly in static, fully connected networks, and impossible in a sparse mobile network, like TurtleNet. Fortunately, in practice, we have found that ignoring these indirect effects does not noticeably degrade performance.

5.4.2 Replication

Network uncertainty can often be masked by replicating the same data over different paths. Replication adds robustness to the network and has been shown to improve delivery rates in disruption-prone environments [5, 75]. Replicating data, if not accounted for, will also unfairly skew rate allocations in much the same way as routing through multiple downstream nodes.

Consider the scenario illustrated in Figure 5.8(a). If n sends all of its data to d and a duplicate copy to d', n might be tempted to send $F_n = 1.0$ to both d_1 and d_2, since each downstream node routed all of its data. In this case, n receives twice its fair allocation.

Instead, a Tula node n incorporates the replication rate in its estimation of the fraction F_n. To this end, n computes F_n as the fraction of its total network transmissions including replicas. In the previous example, n sends $F_n = 0.5$ to both d_1 and d_2, assuming that both paths are favored equally.

5.4.3 Transitive routing

In sparse mobile networks, a node can be several hops away from the sink. In the TurtleNet testbed, for example, some devices were as much as 5 hops away from the base station. In order to deliver data, these nodes must route data transitively as shown in Figure 5.8(c). In the example, nodes v_1 and v_2 route through u_1, while nodes u_1 and u_2 route through n. Assuming energy and connectivity constraints are equal, and n only considers

u_1 and u_2 when allocating resources, n will assign similar rates to u_1 and u_2, even though u_1 is routing 3 nodes' data and u_2 only routes for itself.

Accounting for transitivity requires no change to the COP. Rather the variable, F_i is extended to include upstream traffic. Node u_1 sends the total fraction of traffic that it routes though n, including its own and all its upstream nodes. In the example, if v_1 and v_2 route all their data through u_1, and u_1 in turn routes all its data through n, it will send $F_1 = 3$ to n. The value F_i needs to only be communicated to the immediate downstream neighbor. Nodes v_1 and v_2 communicate their respective values to u_1, and u_1 aggregates the values with its fraction to estimate F_1.

To sum up, the variable F_i represents the fraction of total packets a node sends—including its own and others' packets and accounting for replication—to its downstream neighbor.

5.4.4 Routing through an upstream neighbor

Under most conditions, data packets are routed toward the sink through downstream nodes; however, at times it makes sense to route data to a node that is farther from the destination, as illustrated in Figure 5.8(d). In the example, nodes n and u_1 meet each other frequently, but each rarely meets a shared downstream node d. According to the routing algorithm's distance metric, n is slightly closer to d than u_1. Therefore, n is downstream from u_1, and u_1 will route its data through n. Since u_1 is nearly as likely to meet d as n, node n can significantly increase the probability of delivering data in a timely manner by routing data through u_1 as well. Unfortunately, this is not permitted by our current definition of upstream and downstream nodes. We have observed this scenario often in TurtleNet, and we expect it to occur in any network with social groups and non-uniform mixing.

A solution to this problem is to relax our definition and allow nodes to be both upstream and downstream peers of each other. This solution, however, suffers from the *count-to-*

infinity problem, where a node unknowingly becomes its own downstream peer. This problem can be solved by exchanging link information for all upstream paths, however, it significantly increases Tula's complexity—requiring, in the worst case, that all nodes maintain information about all other nodes in the network.

Therefore, we use a simpler heuristic in Tula, which has worked well in practice. We allow a node to only replicate its own data to upstream peers when appropriate, but disallow forwarding other nodes' data. This simple heuristic helps avoid the *count-to-infinity* problem by ensuring that data is never routed back down the path from which it came.

5.5 Implementation

We have developed two implementations of Tula: a NesC [30] version that runs on a microcontroller platform and a trace-based simulator for repeatable experimentation.

5.5.1 NesC implementation

The goal of the NesC implementation is two-fold. First, it demonstrates that Tula can be implemented in the memory and processor constrained microcontroller platform. Second, it allows us to measure the energy required for each component of Tula—sensing, routing data, solving the COP and exchanging meta data for the routing algorithm. We then instantiate the simulator with real energy measurements. We plan to deploy the full implementation of Tula in our TurtleNet testbed.

The NesC implementation is a fully functioning implementation running on the Shock-Fish TinyNode [16]. The implementation incorporates all of the design features, including the energy/rate allocator, the Eon runtime platform and the Rapid DTN layer. We adapt the Rapid implementation to run on a memory constrained platform. Rapid normally exchanges meta-data about the delay of each packet. Instead, we reduce the meta-data and exchange only the per-node delay and meta-data about a short packet history. We refer to this reduced version as *RapidLite*.

We implemented the allocator in 390 lines of NesC code, and RapidLite in 1172 lines of NesC code. The Eon runtime computes the energy budget of a sensor node by keeping track of the harvested energy and the energy spent for sensing and communication.

5.5.2 Trace-based simulator

Simulation based on real data collected in situ from deployed systems is the most practical method for conducting realistic, fair, and reproducible comparisons between different approaches. Our simulator can take mobility and harvested energy traces from a variety of sources, including traces from our TurtleNet deployment and from UMass DieselNet [10].

The simulator periodically solves the Tula COP and performs sensing and routing based on the rates set by the COP. Connection opportunities are simulated according to the mobility traces. Nodes exchange sensed data and meta information during a connection opportunity. Sensed data is routed based on the RapidLite algorithm. The simulator assigns energy to nodes according to a harvested energy trace. A node accounts for energy consumption due to processing, sensing, and communication using measurements obtained from our implementation.

5.6 Evaluation

Tula adapts sensing and routing rates to provide max-min fairness in the network. We compare the performance of Tula with three different kinds of approaches: *(i) Optimal*, An optimal adaptive policy based on an oracle *(ii) Static policies* that set static sensing and routing rates, and *(iii) Semi-adaptive policies* that either adapt their sensing rate or routing rate, but not both. Our evaluation compares these policies in terms of network performance, energy management, and fairness.

5.6.1 Methodology

We evaluate Tula and alternate policies using the trace-based simulator described in the previous section. In the simulations, each node has 512kB of storage, a 250mAhr battery,

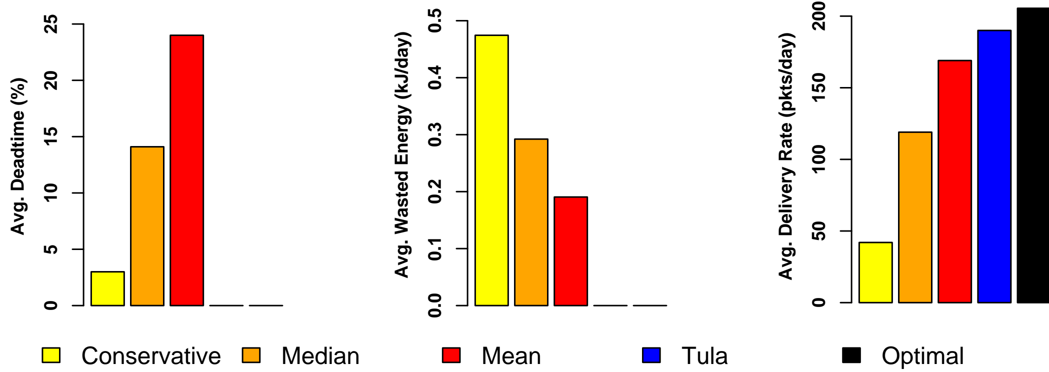


Figure 5.9. Comparison of three static allocation policies, Tula and Optimal. The policies are compared across three metrics: battery dead time, energy wasted since the battery was full and could not charge, and average delivery rate. Tula avoids dead time and wasted energy successfully, and delivers within 92% of the oracle-based optimal policy.

Sensor	Sense/Send ratio
GPS(Max)	2.0×10^{4}
GPS(Avg)	5.0×10^{3}
GPS(Min)	5.0×10^{3}
Accel. (ADXL330)	6.5×10^{-3}
Mag.(HMC1053)	7.2×10^{-1}

Table 5.3. Energy to sense vs. send for common sensors, and the XE1205 low-power radio

and the Tula allocator is run every 2 hours. To evaluate alternate allocation policies, we replace the Tula COP with an allocator that enforces a static, semi-adaptive or optimal allocation policy.

Simulated nodes can be configured to use a variety of sensors. Table 5.3 shows the ratio of sensing and sending cost of three different sensors: GPS, Accelerometer, and Magnetometer.

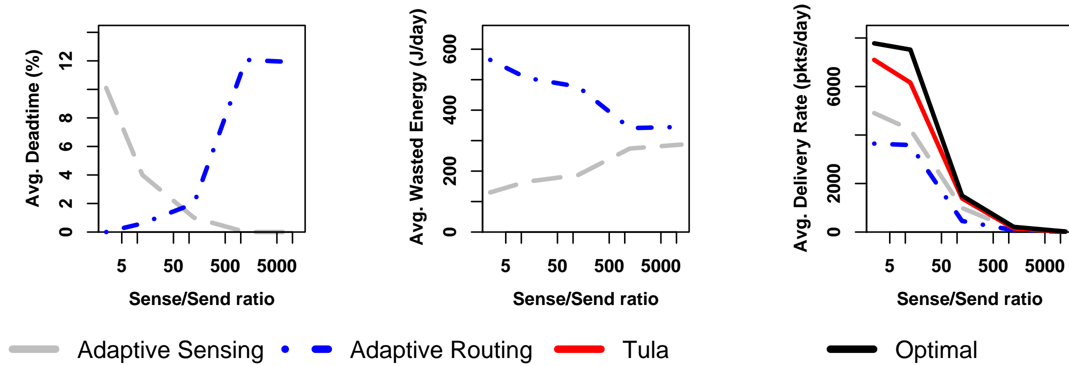


Figure 5.10. Comparison of two semi-adaptive allocation polices, Tula and Optimal. The comparison is performed for different sensor applications with varying sensing to routing ratio.

5.6.1.1 Trace collection

We conduct the trace-based simulations using three traces: TurtleNet, DieselNet and a synthetic mesh trace. The TurtleNet traces include 45 days of data from 17 tracking devices deployed on Gopher tortoises. The data contains measured solar energy, connection opportunities, and the bandwidth available during a connection opportunity. The DieselNet traces are publicly available traces from the UMass DieselNet vehicular network [10] collected from 20 mobile nodes for 55 days in 2008.

The DieselNet bus traces do not contain energy harvesting information, however, we combine historical solar energy traces [2] with the DieselNet bus schedules to estimate the energy harvested at each bus over time. Note that this approach is reasonable only because buses are either parked in the garage (i.e., no energy harvested) or driving on open roads with a clear view of the sky.

Finally, in order to assess the applicability of Tula to fixed networks, we also simulate a synthetic mesh network configuration of 16 nodes arranged in a 4x4 grid topology. We randomly assign energy traces from our TurtleNet data to these mesh nodes.

5.6.1.2 Optimal rate allocation using an oracle

In order to determine the optimal max-min fair rate assignment, we formulate each experimental scenario as a linear program, which can be solved using a general purpose LP solver. Our LP formulation extends the approach used by Fan et al. [24] and we have added support for temporal changes in network connectivity, rate adaptation, and storage limitations, by breaking up the linear program into discrete time segments.

The solver has complete knowledge of future energy harvesting and connectivity for each time segment. The solution from the LP formulation is the maximum max-min fair rate assignment that does not sacrifice node lifetimes or data deliveries. Achieving this rate, in practice, is not feasible since it requires global knowledge of the entire network; however, it provides a useful reference by which to measure system performance.

5.6.2 Network Performance

While Tula adaptively allocates energy for both sensing and routing, there is a wide range of alternative approaches that could be employed. In this section, we compare the performance of Tula with two classes of allocation policies: *Static* and *Semi-adaptive*.

5.6.2.1 Static rate allocation policies

One challenge in designing a static allocation policy is determining what sensing rate should be assigned to the nodes. Due to variation in energy harvesting, setting one sensing rate across all nodes will result in some nodes dying and other nodes having surplus energy. To conduct a fair comparison, we examine a range of behaviors. First, using the oracle-based optimal allocator, we determine the optimal rate allocation for each node. In a real deployment scenario, rate assignments would have to be made based on the system designer's best guess.

We examine the performance of three static rates: *conservative*, a rate that is sustainable by 90% of the nodes in the network; the *median* rate, sustainable by 50% of the nodes; and

the *mean* rate, which can be achieved by only 25% of the nodes. For this experiment, nodes are configured to use the GPS sensor.

The results of this comparison are shown in Figure 5.9. We compare the performance with respect to three metrics: aggregate dead time, total wasted energy, and delivery rate. The aggregate dead time is the total time that nodes in the network have no energy. The total wasted energy is the energy that could not be stored due to limited battery size, even when solar energy was available for harvesting. Dead time is typically a result of over-utilizing energy, while wasted energy is a result of under-utilizing the available energy.

Using the *conservative* rate, nodes are dead only 3% of the time, however, on average nearly 500J of energy—enough to collect nearly 200 sensor readings—are wasted daily per node. The *mean* rate wastes much less energy, but on average nodes are dead for 25% of the time. The *median* rate provides an unsatisfying tradeoff between the two extremes, resulting in mediocre performance across all three metrics.

The results show that in a network with wide variations in energy availability and connectivity, a static scheme will perform poorly, regardless of the rate that is assigned. In contrast, by adapting per-node sensing rates, Tula is able to *completely* avoid dead time and wasted energy, resulting in 11% more data collected than using the mean rate, and within 92% of the optimal result.

5.6.2.2 Semi-adaptive rate allocation policies

Next, we make a similar comparison with two alternate adaptive policies—a policy that adapts only the routing rate or only the sensing rates.

In the *adaptive sensing* policy, routing decisions are made without any energy restrictions. However, the node adapts its sensing rates according to the remaining energy. In contrast, in the *adaptive routing* policy, sensing rates are fixed, and routing decisions are made adaptively using the energy that remains after sensing. The adaptive routing policy requires a fixed sensing rate, and we set the rate to the *conservative* rate described previ-

ously. Recall that the *conservative* rate is a rate sustainable by 90% of the nodes. Setting the static rate to other values results in similar or worse trade-offs.

Unlike the static allocation, the performance of partially adaptive rate allocation depends on the sensor. For example, if nodes only obtain accelerometer readings, the sensing cost is low enough that adapting the sensing rate does not provide benefits. Alternatively, if the sensor application obtains GPS readings, adapting the sensing rate is important to ensure that a node does not exhaust its battery. Accordingly, we compare the performance of the different allocation policies for a range of sensors with varying energy requirements (as shown in Table 5.3).

Figure 5.10, illustrates the chief shortcoming of these partially adaptive approaches—their ability to adapt to changes is limited by the consumption of the static tasks. As the energy for sensing increases (left of the graph), the dead time when using adaptive routing policy increases from 0 to 12%. On the other hand, when the energy for sensing is low, more packets are sensed and routed. As a result, the average dead time of the adaptive sensing policy increases to 10%.

Both the adaptive sensing and adaptive routing policy waste between 200 J to 600 J daily depending on the policy and the sensor. In contrast, Tula optimizes both sensing and routing and the policy incurs no dead time and no wasted energy. In terms of delivery rate, Tula collects on average 30-50% more data than both the semi-adaptive techniques.

5.6.2.3 Network performance over DieselNet and Mesh

Figure 5.11 shows the delivery rates achieved by Tula for three different network configurations. Tula achieves a delivery rate of within 75% of the Optimal policy over TurtleNet and DieselNet, even without future knowledge of the harvest energy or node meeting schedules.

On a static mesh network, Tula is able to sense and deliver data within 85% of the Optimal policy for a range of sensors, showing that Tula can adapt well to different topolo-

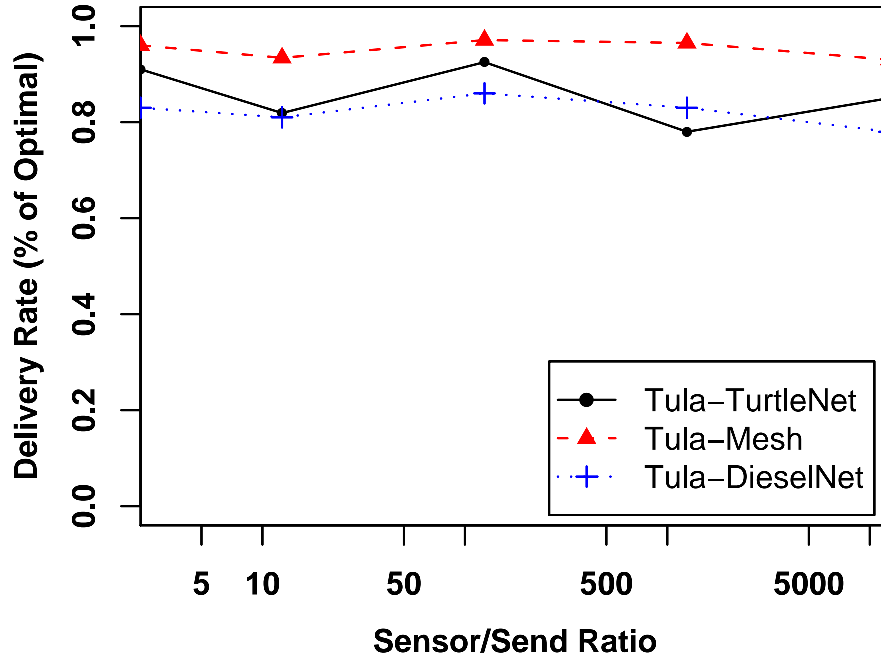


Figure 5.11. Delivery rate of Tula normalized to the optimal delivery rate over three networks configurations: TurtleNet, a static 4x4 grid mesh network, and the DieselNet vehicular traces.

gies. More importantly, in the absence of mobility, the rate set by the Tula allocation policy converges close to the optimal rate.

5.6.3 Fairness

The objective of the Tula allocation policy is to set rates such that data is sensed and delivered to the sink at a max-min fair rate. In this section, we evaluate the fairness of Tula using two metrics. First, we compare the per-node delivery rate of Tula with Optimal. Recall that the optimal oracle-based rate allocator is also designed to set max-min fair rates.

Figures 5.12, 5.13 and 5.14 show the per-node delivery rate of Tula compared to optimal for the three network configurations: TurtleNet, DieselNet and Mesh, respectively. For all three networks, the per-node delivery rate of Tula is close to the optimal per-node

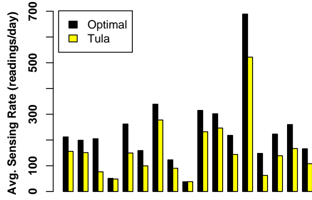


Figure 5.12. TurtleNet traces: Average per-node delivery rate

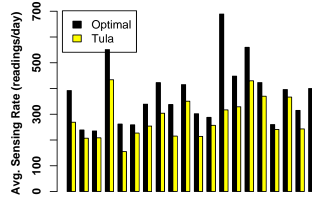


Figure 5.13. DieselNet traces: Average per-node delivery rate

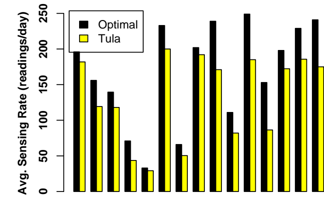


Figure 5.14. Mesh: Average per-node delivery rate

rate. For example, in TurtleNet, nearly all of the nodes achieve a delivery rate within 75% of the optimal. Similar performance is seen for both Mesh and DieselNet.

5.6.4 Overhead

Finally, we quantify the overhead of Tula using measurements from our TinyNode-based implementation. Current draw is measured using a NI-PCI 6251 DAQ, measuring the voltage drop across a low-tolerance current-sense resistor. Simultaneously measuring the voltage across the device's battery allows us to compute the energy consumption. The measurements are shown in Table 5.4.

Apart from the core sensing and networking tasks, energy is incurred when periodically solving the Tula COP and computing the device's energy budget. However, both tasks only consume roughly as much energy as transmitting 2-3 packets over the device's radio. In addition to energy costs, our implementation of Tula requires 1.5kB of RAM and 22kB of additional program space in addition to the space requirements of the Eon runtime system. These size requirements are easily met by nearly all microcontroller-based platforms that are currently in use.

Tula Energy/Time Overhead

Operation	Energy	Time
Solve COP	$0.9 - 2.3$mJ	$0.5 - 1.35$s
Compute Energy Budget	$1.4 - 1.8$mJ	$0.8 - 1.0$ms

Memory Overhead

RAM overhead	1.5kB
Additional code size	22kB

Table 5.4. Measurements of Tula overhead.

5.7 Related Work

Tula builds on a large body of previous work in several fields: challenged networking, rate allocation and fairness, mobile sensor networks, and adaptation. In many ways this synthesis is too large to cover here, so we provide the most relevant work.

5.7.1 Mobile sensor networks

Most sensor network papers in the literature are static and connected; however, some previous mobile sensor deployments have shared many of the same goals as Tula. ZebraNet [88], for example, initially explored the use of in-situ sensing devices for wildlife tracking. These first devices were large (>0.5kg) and masked energy variations with large batteries and solar panels—too large for most animals to carry; however, they set the stage for future mobile sensing systems, like Tula.

Of course, mobile sensor systems are not limited to wildlife tracking. The CarTel [38] and Pothole Patrol [20] projects used mobile sensors in vehicles to identify traffic congestion and to provide cities with valuable road-quality information. Like most vehicle-based networks, these devices receive power from the vehicles. Several projects have also used mobile phones as sensing platforms [19, 42, 54, 80]. While some systems make efforts to conserve energy, all rely heavily on the user to manage the phone's energy supply.

5.7.2 Low power sensor networks

Energy scarcity is a first class design concern for wireless sensor networks. Low-power hardware platforms with energy harvesting support [41, 49, 74] as well as algorithms for estimating and predicting energy harvesting and consumption [17, 44, 74] are crucial components of all perpetual systems. Additionally, a variety of energy-aware networking techniques have been proposed for use with low-power sensors, including energy-aware clustering [86], aggregation, and traffic shaping to extend device lifetimes [71]. However, previous research on energy aware sensor networks has focused on static network topologies that improve node lifetime, but do not focus on perpetual operation.

5.7.3 Challenged networks

A wealth of previous research has focused on building disruption tolerant networks with sparse connectivity. Research in this area has provided a range of protocols [5, 39, 50, 75], which opportunistically forward and replicate packets to mobile peers. Most DTN solutions have targeted vehicular [11] and personal device networks [37]. Whether tapping into a vehicle's battery or relying on user-facilitated recharges, previous solutions assume a steady and unlimited energy supply, and neglect the challenge of energy scarcity which is central to any untethered system.

5.7.4 Fair network rate allocation

A variety of fairness policies have been proposed [8, 46], along with many techniques for enforcing those policies in wireless networks [28,78,91]. Of these approaches, the most closely related work provides both centralized and distributed algorithms for enforcing max-min fairness in networks that have rechargeable sensors [24,51]. However, the authors assume that the routes in the network are static, and that the energy profile of a node is known in advance. Tula enforces max-min fairness in networks with unpredictable network connectivity and dynamically changing energy constraints.

Another closely related are the Peloton [82] and IDEA [3], which have similar goals and have been developed concurrently with Tula. Both are efforts to extend Pixie [53] to support for network-wide cooperation, just as Tula extends Eon. Despite their similarities, IDEA and Tula have important differences, that parallel the differences between Pixie [53] and Eon [74]. In IDEA, nodes exchange information about battery fullness, charging characteristics, and system energy load, along with utility functions that describe the value of system states. Like Pixie, this reflects IDEA's focus on maximizing resource visibility and a designers possibilities. In contrast, Tula focuses on making the complications of in-network cooperation invisible and automatic. Tula also automatically balances inherently dependent sensing and routing tasks—a dependence which is not addressed by IDEA.

Finally, myriad of techniques aim at improving performance and enabling new applications by providing additional coordination across traditionally independent network layers [15, 77], when legacy abstractions fail to meet the needs of emerging systems and environments. Tula is also a cross-layer approach providing a tight link between the application and network in order to address the combined challenges of mobility, heterogeneity and perpetual operation.

5.8 Discussion

In this chapter, we have described Tula, a system which balances sensing with packet delivery for energy harvesting mobile sensor networks. Tula represents a first step in managing the combined resources of a network of constrained nodes, balancing sensing and communication, while maintaining a cooperative system for delivering data. Our evaluation of Tula, using mobility and energy traces from our TurtleNet deployment, shows that Tula collects and delivers data within 75% of an optimal oracular policy. In addition, we have shown that Tula successfully enforces a max-min fairness policy and is suitable for use on low power sensing platforms.

CHAPTER 6

CONCLUSIONS AND FUTURE WORK

This chapter describes our conclusions about both Eon and Tula, as well as the limitations of our work and planned future directions.

6.1 Conclusion

Perpetual systems promise to transform our understanding of the natural world and to extend the benefits of computing and sensing technologies into challenging environments, remote locations, and at an unprecedented scale. Unfortunately, their many deployment challenges (e.g., uncertainty and variation) make building and tuning perpetual systems difficult for experts and impossible for less technical users.

This dissertation describes techniques that overcome this tension allowing system designers to build systems that operate perpetually and deliver data efficiently and fairly, without sacrificing simplicity. This is accomplished in two ways: first we use Eon, the first energy-aware programming language, which allows programmers to simply express application specific energy policies and then delegate the complexities of energy-aware adaptation to the underlying system; and second, we use Tula to balance data collection with data delivery and ensure that each device has fair access to network resources.

We have shown that Eon can reduce program development time by up to 4X, and that the Eon runtime system's simple adaptation algorithms are able to effectively adjust program behavior, extend device lifetime, and avoid wasting energy—performing on par with a hypothetical oracle-based system using perfect energy prediction. In addition to our

quantitative results, in our own experience using Eon, we have found the language to be expressive, simple, and flexible.

Tula brings together the work of two separate areas of research—adaptive sensing and disruption-tolerant networking—allowing the inherently dependent activities of data gathering or production and data delivery to be coordinated automatically. We have also shown that Tula successfully ensures that all nodes have fair access to network resources, assigning rates across the network within 75% of the optimal max-min fair rate allocation.

6.2 Future Work

While our work lays the groundwork for building perpetual mobile systems, there are many directions in which this research could be extended.

Other Network Models: We have, so far, focused on mobile tracking systems, which collect data over time and forward that data to a collection point or base station. This model for sensing and routing is common and straightforward; however, there are other models that are used by sensor network applications, including query-based collection [55], and in-network aggregation [23].

Incorporating these alternative models in Tula, requires the system to estimate the effect that aggregation and querying have on network load. In the case of querying, network load and the flexibility with which Tula can adapt would also depend on the nature of the query itself. Supporting aggregation would require the system to estimate the amount of compression achieved by aggregation at each hop in the network.

Prioritized Network Flows: Sensor network applications may also collect data from multiple different types of sensors, and some data may have higher priority than others. A sensor network monitoring a volcano [83], for example, may want to report dangerous events immediately, while those that are merely scientifically interesting can be collected after the danger passes.

In general, data prioritization is complementary to this work. Eon already assigns priority to program flows based on the programmer's annotations, and these assignments could naturally translate into network priorities. Accounting for packet priorities in Tula would naturally complicate the optimization problem, but the extension is likely to be straightforward.

In this dissertation we have chosen to focus on the fundamental challenges that face perpetual systems. In order to maintain this focus, these features, extensions, and models, while important, are left as future work—being too intricate to cover in a single dissertation.

6.3 The Final Word

We argue that as the scale and complexity of mobile sensing systems increases, successful systems will be adaptive and self managing—capable of long deployments in a wide range of environments and adverse conditions. These perpetual systems will only see widespread use when they are simple to program, optimize, and understand, while leveraging proven techniques for estimating, predicting, and efficiently sharing network and energy resources. Our work demonstrates that these goals are within reach for today's energy constrained microcontroller-based platforms, and provides the research community with the techniques and tools necessary to achieve them.

BIBLIOGRAPHY

[1] http://prisms.cs.umass.edu/dome.

[2] http://weather.cs.umass.edu/.

[3] IDEA: integrated distributed energy awareness for wireless sensor networks. In *Proceedings of the 8th Annual International Conference on Mobile Systems, Applications and Services (MobiSys '10)* (San Francisco, CA, USA, June 2010).

[4] Anand, M., Nightingale, E. B., and Flinn, J. Self-tuning wireless network power management. In *Proceedings of the 9th ACM International Conference on Mobile Computing and Networking (MobiCom'03)* (San Diego, CA, September 2003).

[5] Balasubramanian, Aruna, Levine, Brian Neil, and Venkataramani, Arun. DTN Routing as a Resource Allocation Problem. In *Proc. ACM Sigcomm* (August 2007).

[6] Balasubramanian, Aruna, Mahajan, Ratul, Venkataramani, Arun, Levine, Brian, and Zahorjan, John. Interactive WiFi Connectivity for Moving Vehicles. In *ACM SIGCOMM* (August 2008).

[7] Berry, G., and Gonthier, G. The ESTEREL synchronous programming language: design, semantics, implementation. *Science of Computer Programming 19*, 2 (November 1992), 87–152.

[8] Boudec, Jean-Yves. Rate adaptation, congestion control and fairness: A tutorial, 2000.

[9] Burgess, John, Bissias, George, Corner, Mark D., and Levine, Brian Neil. Surviving Attacks on Disruption-Tolerant Networks without Authentication. In *Proc. ACM Mobihoc* (Montreal, Quebec, Canada, September 2007).

[10] Burgess, John, Gallagher, Brian, Jensen, David, and Levine, Brian Neil. MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks. In *IEEE INFOCOM* (April 2006).

[11] Burgess, John, Gallagher, Brian, Jensen, David, and Levine, Brian Neil. MaxProp: Routing for Vehicle-Based Disruption-Tolerant Networks. In *Proc. IEEE INFOCOM* (April 2006).

[12] Burns, Brendan, Grimaldi, Kevin, Kostadinov, Alexander, Berger, Emery D., and Corner, Mark D. Flux: A language for programming high-performance servers. In *Proc. USENIX Annual Technical Conference* (May 2006).

[13] Chen, Wei-Peng, and Sha, Lui. An energy-aware data-centric generic utility based approach in wireless sensor networks. In *Proceedings of the third international symposium on Information processing in sensor networks (IPSN)* (April 2004), pp. 215 – 224.

[14] Chinn, G., Desai, S., DiStefano, Eric, Ravichandran, K., and Thakkar, S. Mobile PC platforms enabled with Intel Centrino mobile technology. *Intel Technology Journal 7*, 2 (May 2003).

[15] Conti, M., Maselli, G., Turi, G., and Giordano, S. Cross-layering in mobile ad hoc network design. *Computer 37*, 2 (Feb 2004), 48–51.

[16] Dubois-Ferriere, Henri, Meier, Roger, Fabre, Laurent, and Metrailler, Pierre. TinyNode: A comprehensive platform for wireless sensor network applications. In *Proceedings of the fifth international conference on Information processing in sensor networks (Poster)* (Nashville, TN, USA, April 2006), pp. 358–365.

[17] Dunkels, Adam, Osterlind, Fredrik, Tsiftes, Nicolas, and He, Zhitao. Software-based on-line energy estimation for sensor nodes. In *EmNets '07: Proceedings of the 4th workshop on Embedded networked sensors* (New York, NY, USA, 2007), ACM, pp. 28–32.

[18] Dutta, Prabal, Feldmeier, Mark, Paradiso, Joseph, and Culler, David. Energy metering for free: Augmenting switching regulators for real-time monitoring. In *IPSN '08: Proceedings of the 7th international conference on Information processing in sensor networks* (Washington, DC, USA, 2008), IEEE Computer Society, pp. 283–294.

[19] Eagle, Nathan, and (Sandy) Pentland, Alex. Reality mining: sensing complex social systems. *Personal Ubiquitous Computing 10*, 4 (2006), 255–268.

[20] Eriksson, Jakob, Girod, Lewis, Hull, Bret, Newton, Ryan, Madden, Samuel, and Balakrishnan, Hari. The pothole patrol: using a mobile sensor network for road surface monitoring. In *MobiSys '08: Proceeding of the 6th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2008), ACM, pp. 29–39.

[21] Ernst, C. H., Lovich, J. E., and Barbour, R. W. *Turtles of the United States and Canada*. Smithsonian Institute Press, 1994.

[22] et. al., John Porter. Wireless sensor networks for ecology. *BioScience 55*, 7 (July 2005).

[23] Fan, Kai-Wei, Liu, Sha, and Sinha, Prasun. Structure-free data aggregation in sensor networks. *IEEE Transactions on Mobile Computing 6*, 8 (2007), 929–942.

[24] Fan, Kai-Wei, Zheng, Zizhan, and Sinha, Prasun. Steady and fair rate allocation for rechargeable sensors in perpetual sensor networks. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems* (New York, NY, USA, 2008), ACM, pp. 239–252.

[25] Fioravanti-Score, A., Mitchell, Sarah V., and Williamson, J. Michael. Use of Satellite Telemetry Technology to Enhance Research and Education in the Protection of Loggerhead Sea Turtles. In *19th Annual Symposium on Sea Turtle Biology and Conservation* (1999).

[26] Flinn, J., and Satyanarayanan, M. Powerscope: A tool for profiling the energy usage of mobile applications. In *Proceedings of the Second IEEE Workshop on Mobile Computer Systems and Applications* (New Orleans, LA, February 1999).

[27] Flinn, J., and Satyanarayanan, M. Managing battery lifetime with energy-aware adaptation. *ACM Transactions on Computer Systems (TOCS) 22*, 2 (May 2004).

[28] Gambiroza, Violeta, Sadeghi, Bahareh, and Knightly, Edward W. End-to-end performance and fairness in multihop wireless backhaul networks. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking* (New York, NY, USA, 2004), ACM, pp. 287–301.

[29] Gay, D., Levis, P., Behren, R. V., Welsh, M., Brewer, E., and Culler, D. The nesC language: A holistic approach to networked embedded systems. In *Proceedings of Programming Language Design and Implementation (PLDI)* (June 2003).

[30] Gay, David, Levis, Philip, von Behren, Robert, Welsh, Matt, Brewer, Eric, and Culler, David. The nesc language: A holistic approach to networked embedded systems. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation* (New York, NY, USA, 2003), ACM, pp. 1–11.

[31] Gelernter, David, and Carriero, Nicholas. Coordination languages and their significance. *Commun. ACM 35*, 2 (1992), 96.

[32] Greenstein, Ben, Kohler, Eddie, and Estrin, Deborah. A sensor network application construction kit (snack). In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (New York, NY, USA, 2004), ACM Press, pp. 69–80.

[33] Gu, Yu, Zhu, Ting, and He, Tian. ESC: Energy Synchronized Communication in Sustainable Sensor Networks. In *Proceedings of the 17th International Conference on Network Protocols (ICNP '09)* (October 2009).

[34] Gummadi, Ramakrishna, Kothari, Nupur, Govindan, Ramesh, and Millstein, Todd. Kairos: a macro-programming system for wireless sensor networks. In *SOSP '05: Proceedings of the twentieth ACM symposium on Operating systems principles* (New York, NY, USA, 2005), ACM, pp. 1–2.

[35] Halbwachs, N., Caspi, P., Raymond, P., and Pilaud, D. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE 79*, 9 (September 1991), 1305–1320.

[36] Hartung, C., Holbrook, S., Han, R., and Seielstad, C. olbrook, r. han, c. seielstad, “firewxnet: A multi-tiered portable wireless system for monitoring weather conditions in wildland fire environments. In *Fourth International Conference on Mobile Systems, Applications and Services (MobiSys)* (2006), pp. 28–41.

[37] Hui, Pan, Chaintreau, Augustin, Scott, James, Gass, Richard, Crowcroft, Jon, and Diot, Christophe. Pocket Switched Networks and Human Mobility in Conference Environments. In *Proc. ACM Workshop on Delay-Tolerant Networking* (Aug. 2005), pp. 244–251.

[38] Hull, Bret, Bychkovsky, Vladimir, Zhang, Yang, Chen, Kevin, Goraczko, Michel, Miu, Allen, Shih, Eugene, Balakrishnan, Hari, and Madden, Samuel. CarTel: A Distributed Mobile Sensor Computing System. In *ACM SenSys* (October 2006), pp. 125–138.

[39] Jain, S., Fall, K., and Patra, R. Routing in a Delay Tolerant Network. In *Proc. ACM SIGCOMM* (August 2004), pp. 145–158.

[40] Jiang, Xiaofan, Dutta, Prabal, Culler, David, and Stoica, Ion. Micro power meter for energy monitoring of wireless sensor networks at scale. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM, pp. 186–195.

[41] Jiang, Xiaofan, Polastre, Joseph, and Culler, David E. Perpetual environmentally powered sensor networks. In *IPSN* (2005), pp. 463–468.

[42] Kansal, Aman, Goraczko, Michel, and Zhao, Feng. Building a sensor network of mobile phones. In *IPSN '07: Proceedings of the 6th international conference on Information processing in sensor networks* (New York, NY, USA, 2007), ACM, pp. 547–548.

[43] Kansal, Aman, Hsu, Jason, Srivastava, Mani B, and Raghunathan, Vijay. Harvesting aware power management for sensor networks. In *43rd Design Automation Conference (DAC)* (July 2006).

[44] Kansal, Aman, Hsu, Jason, Zahedi, Sadaf, and Srivastava, Mani B. Power management in energy harvesting sensor networks. *ACM Transactions on Embedded Computing Systems* (May 2006).

[45] Kulkarni, P., Ganesan, D., and Shenoy, P. Senseye: A multi-tier camera sensor network. In *ACM Multimedia* (2005).

[46] Kushner, H.J., and Whiting, P.A. Convergence of proportional-fair sharing algorithms under general conditions. *Wireless Communications, IEEE Transactions on 3*, 4 (July 2004), 1250–1259.

[47] Lachenmann, Andreas, Marrón, Pedro José, Minder, Daniel, and Rothermel, Kurt. Meeting lifetime goals with energy levels. In *Proc. of the 5th ACM Conference on Embedded Networked Sensor Systems* (2007), pp. 131–144.

[48] Landsiedel, O., Wehrle, K., and Gotz, S. Accurate prediction of power consumption in sensor networks. In *EmNets '05: Proceedings of the 2nd IEEE workshop on Embedded Networked Sensors* (Washington, DC, USA, 2005), IEEE Computer Society, pp. 37–44.

[49] Lin, Kris, Hsu, Jason, Zahedi, Sadaf, Lee, David C, Friedman, Jonathan, Kansal, Aman, Raghunathan, Vijay, and Srivastava, Mani B. Heliomote: Enabling long-lived sensor networks through solar energy harvesting. In *Proceedings of ACM Sensys* (November 2005).

[50] Lindgren, A., Doria, A., and Scheln, O. Probabilistic Routing in Intermittently Connected Networks. In *Proc. Workshop on Service Assurance with Partial and Intermittent Resources* (August 2004).

[51] Liu, Ren-Shiou, Sinha, Prasun, and Koksal, Can Emre. Joint energy management and resource allocation in rechargeable sensor networks. In *Proceedings of IEEE Infocom* (2010).

[52] Lorch, J., and Smith, A. J. Energy consumption of apple macintosh computers. *IEEE Micro 18*, 6 (November/December 1998).

[53] Lorincz, Konrad, Chen, Bor-rong, Waterman, Jason, Werner-Allen, Geoff, and Welsh, Matt. Resource aware programming in the pixie os. In *SenSys '08: Proceedings of the 6th ACM conference on Embedded network sensor systems* (New York, NY, USA, 2008), ACM, pp. 211–224.

[54] Lu, Hong, Pan, Wei, Lane, Nicholas D., Choudhury, Tanzeem, and Campbell, Andrew T. Soundsense: scalable sound sensing for people-centric applications on mobile phones. In *MobiSys '09: Proceedings of the 7th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2009), ACM, pp. 165–178.

[55] Madden, Samuel R., Franklin, Michael J., Hellerstein, Joseph M., and Hong, Wei. Tinydb: an acquisitional query processing system for sensor networks. *ACM Trans. Database Syst. 30*, 1 (2005), 122–173.

[56] Mainland, Geoff, Parkes, David C., and Welsh, Matt. Decentralized, adaptive resource allocation for sensor networks. In *Proceedings of the 2nd USENIX/ACM Symposium on Networked Systems Design and Implementation (NSDI 2005)* (May 2005).

[57] Mainland, Geoffrey, Welsh, Matt, and Morrisett, Greg. Flask: A language for data-driven sensor network programs. Tech. Rep. TR-13-06, Harvard University, Division of Engineering and Applied Sciences, May 2006.

[58] Mainwaring, A., Polastre, J., Szewczyk, R., Culler, D., and Anderson, J. Wireless sensor networks for habitat monitoring. In *Workshop on Wireless Sensor Networks and Applications* (Atlanta, GA, September 2002).

[59] Meninger, S., Mur-Miranda, J. O., Amirtharajah, R., Chandrakasan, A., and Lang, J. H. Vibration-to-electric energy conversion. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems 9*, 1 (February 2001), 64–76.

[60] Morrison, J. Paul. *Flow-Based Programming: A new approach to application development*. Van Nostrand Reinhold, 1994.

[61] Nellis, David W., and Small, Vonnie. Mongoose predation on sea turtle eggs and nests. *Biotropica 15*, 2 (1983), 159–160.

[62] Nemmaluri, Aditya, Corner, Mark D., and Shenoy, Prashant. Sherlock: Automatically locating objects for humans. In *Proceedings of Mobisys* (Breckenridge, CO, June 2008).

[63] Neugebauer, Rolf, and McAuley, Derek. Energy is just another resource: Energy accounting and energy pricing in the nemesis os. In *Proceedings of the 8th IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII)* (Schloss Elmau, Germany, May 2001).

[64] Newton, Ryan, Arvind, and Welsh, Matt. Building up to macroprogramming: an intermediate language for sensor networks. In *IPSN '05: Proceedings of the 4th international symposium on Information processing in sensor networks* (Piscataway, NJ, USA, 2005), IEEE Press, p. 6.

[65] Noble, B., Satyanarayanan, M., Narayanan, D., Tilton, J.E., Flinn, J., and Walker, K. Agile application-aware adaptation for mobility. In *Proceedings of the 16th ACM Symposium on Operating System Principles* (St. Malo, France, October 1997).

[66] Polastre, J., Szewczyk, R., and Culler, D. Telos: Enabling ultra-low power wireless research. In *Proc. IPSN/SPOTS* (April 2005).

[67] Polastre, J., Szewczyk, R., Sharp, C., , and Culler, D. The mote revolution: Low power wireless sensor networks. In *Proceedings of the 16th Symposium on High Performance Chips (HotChips)* (August 2004).

[68] Polastre, Joseph, Hill, Jason, and Culler, David E. Versatile low power media access for wireless sensor networks. In *SenSys* (2004), pp. 95–107.

[69] Priya, Shashank, Chen, Chih-Ta, Fye, Darren, and Zahnd, Jeff. Piezoelectric windmill: A novel solution to remote sensing. *Japanese Journal of Applied Physics 44*, 3 (2005), 104–107.

[70] Rowe, Anthony, Rosenberg, Charles, and Nourbakhsh, Illah. A low cost embedded color vision system. In *Proceedings of Intelligent Robots and System* (EPFL Switzerland s, September 2002), pp. 208–213.

[71] Schurgers, C., and Srivastava, M.B. Energy efficient routing in wireless sensor networks. *Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force. IEEE 1* (2001), 357–361 vol.1.

[72] Shah, Rahul C., and Rabaey, Jan M. Energy aware routing for low energy ad hoc sensor networks, 2002.

[73] Shnayder, Victor, Hempstead, Mark, Chen, Bor-Rong, and Welsh, Matt. Powertossim: Efficient power simulation for tinyos applications. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)* (2004).

[74] Sorber, Jacob, Kostadinov, Alexander, Garber, Matthew, Brennan, Matthew, Corner, Mark D., and Berger, Emery D. Eon: A Language and Runtime System for Perpetual Systems. In *Proc. ACM SenSys* (Sydney, Australia, November 2007).

[75] Spyropoulos, Thrasyvoulos, Psounis, Konstantinos, and Raghavendra, Cauligi. Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks. In *Proc. ACM Workshop on Delay-Tolerant Networking* (Aug. 2005), pp. 252–259.

[76] Spyropoulos, Thrasyvoulos, Psounis, Konstantinos, and Raghavendra, Cauligi S. Spray and Wait: An Efficient Routing Scheme for Intermittently Connected Mobile Networks. In *Proc. ACM WDTN* (Aug. 2005), pp. 252–259.

[77] Srivastava, V., and Motani, M. Cross-layer design: a survey and the road ahead. *Communications Magazine, IEEE 43*, 12 (Dec. 2005), 112–119.

[78] Tassiulas, L., and Sarkar, S. Maxmin fair scheduling in wireless networks. *INFOCOM 2002. Twenty-First Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE 2* (2002), 763–772 vol.2.

[79] Vicaire, Pascal A., Xie, Zhiheng, Hoque, Enamul, and Stankovic, John A. Physicalnet: A generic framework for managing and programming across pervasive computing networks. *Real-Time and Embedded Technology and Applications Symposium, IEEE 0* (2010), 269–278.

[80] Wang, Yi, Lin, Jialiu, Annavaram, Murali, Jacobson, Quinn A., Hong, Jason, Krishnamachari, Bhaskar, and Sadeh, Norman. A framework of energy efficient mobile sensing for automatic user state recognition. In *MobiSys '09: Proceedings of the 7th international conference on Mobile systems, applications, and services* (New York, NY, USA, 2009), ACM, pp. 179–192.

[81] Want, R., Pering, T., Danneels, G., Kumar, M., Sundar, M., and Light, J. The Personal Server - Changing the way we think about ubiquitous computing. In *Proceedings of Ubicomp 2002: 4th International Conference on Ubiquitous Computing* (Goteborg, Sweden, September 2002).

[82] Waterman, Jason, Challen, Geoffrey Werner, and Welsh, Matt. Peloton: Coordinated resource management for sensor networks. In *Proceedings of the 12th Workshop on Hot Topics in Operating Systems (HotOS '09)* (May 2009).

[83] Werner-Allen, Geoff, Lorincz, Konrad, Johnson, Jeff, Lees, Jonathan, and Welsh, Matt. Fidelity and yield in a volcano monitoring sensor network. In *Proceedings of the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI 2006)* (Seattle, WA, November 2006).

[84] Xiao, Shuo, Sivaraman, Vijay, and Burdett, Alison. Adapting radio transmit power in wireless body area sensor networks. In *BodyNets '08: Proceedings of the ICST 3rd international conference on Body area networks* (ICST, Brussels, Belgium, Belgium, 2008), ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), pp. 1–8.

[85] Ye, Wei, Heidemann, John, and Estrin, Deborah. An energy-efficient mac protocol for wireless sensor networks. In *21st International Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM)* (New York, NY, USA, June 2002), pp. 1567–1576.

[86] Younis, O., and Fahmy, S. HEED: A hybrid, energy-efficient, distributed clustering approach for ad-hoc sensor networks. *IEEE Transactions on Mobile Computing 4*, 4 (October 2004).

[87] Zeng, H., Ellis, C. S., Lebeck, A. R., and Vahdat, A. ECOSystem: Managing energy as a first class operating system resource. In *Proceedings of the Tenth international conference on architectural support for programming languages and operating systems* (San Jose, CA, October 2002).

[88] Zhang, Pei, Sadler, Christopher M., Lyon, Stephen A., and Martonosi, Margaret. Hardware design experiences in zebranet. In *SenSys '04: Proceedings of the 2nd international conference on Embedded networked sensor systems* (New York, NY, USA, 2004), ACM, pp. 227–238.

[89] Zhang, Xiaolan, Kurose, Jim, Levine, Brian Neil, Towsley, Don, and Zhang, Honggang. Study of a Bus-Based Disruption Tolerant Network: Mobility Modeling and Impact on Routing. In *Proc. ACM Mobicom* (September 2007).

[90] Zhao, Wenrui, Chen, Yang, Ammar, Mostafa, Corner, Mark D., Levine, Brian Neil, and Zegura, Ellen. Capacity Enhancement using Throwboxes in DTNs. In *Proc. IEEE Intl Conf on Mobile Ad hoc and Sensor Systems (MASS)* (Oct 2006), pp. 31–40.

[91] Zhu, Junhua, Hung, Ka-Lok, and Bensaou, Brahim. Tradeoff between network lifetime and fair rate allocation in wireless sensor networks with multi-path routing. In *MSWiM '06: Proceedings of the 9th ACM international symposium on Modeling analysis and simulation of wireless and mobile systems* (New York, NY, USA, 2006), ACM, pp. 301–308.