

12-1-2008

# A hierarchical group model for programming sensor networks

James Horey

Follow this and additional works at: [https://digitalrepository.unm.edu/cs\\_etds](https://digitalrepository.unm.edu/cs_etds)

---

## Recommended Citation

Horey, James. "A hierarchical group model for programming sensor networks." (2008). [https://digitalrepository.unm.edu/cs\\_etds/2](https://digitalrepository.unm.edu/cs_etds/2)

This Dissertation is brought to you for free and open access by the Engineering ETDs at UNM Digital Repository. It has been accepted for inclusion in Computer Science ETDs by an authorized administrator of UNM Digital Repository. For more information, please contact [disc@unm.edu](mailto:disc@unm.edu).

James Horey

*Candidate*

Computer Science

*Department*

This dissertation is approved, and it is acceptable in quality and form for publication on microfilm:

*Approved by the Dissertation Committee:*

\_\_\_\_\_, Chairperson

Accepted:

\_\_\_\_\_  
*Dean, Graduate School*

\_\_\_\_\_  
*Date*

# **A Hierarchical Group Model for Programming Sensor Networks**

by

**James Horey**

B.A., Computer Science, Hendrix College, 2003

DISSERTATION

Submitted in Partial Fulfillment of the  
Requirements for the Degree of

Doctor of Philosophy  
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

©2008, James Horey

# Dedication

*I dedicate this dissertation to my family; past, present, and future.*

# Acknowledgments

This dissertation would not have been possible without the mentorship of my advisor Arthur B. Maccabe. He has provided invaluable support as a teacher, research advisor, and friend. Our early conversations have inspired and influenced much of this work. I'd also like to thank Stephanie Forrest for our fruitful collaboration and her willingness to include me in her research activities. I also like to acknowledge the other members of my committee, Patrick Bridges and Wennie Shu for their feedback and support.

I feel privileged to have been a member of the Scalable Systems Laboratory at the University of New Mexico. My labmates have often provided both academic support and friendship. My wife, Alison Boyer, has continually encouraged me to do my best, and I owe her more thanks than can be expressed in words.

Finally, I'd like to acknowledge my funding sources, Los Alamos National Laboratory and the National Nuclear Security Agency.

# **A Hierarchical Group Model for Programming Sensor Networks**

by

**James Horey**

ABSTRACT OF DISSERTATION

Submitted in Partial Fulfillment of the  
Requirements for the Degree of

Doctor of Philosophy  
Computer Science

The University of New Mexico

Albuquerque, New Mexico

December, 2008

# **A Hierarchical Group Model for Programming Sensor Networks**

by

**James Horey**

B.A., Computer Science, Hendrix College, 2003

Ph.D., Computer Science, University of New Mexico, 2008

## **Abstract**

A hierarchical group model that decouples computation from hardware can characterize and aid in the construction of sensor network software with minimal overhead. Future sensor network applications will move beyond static, homogeneous deployments to include dynamic, heterogeneous elements. These sensor networks will also gain new users, including casual users who will expect intuitive interfaces to interact with sensor networks. To address these challenges, a new computational model and a system implementing the model are presented. This model ensures that computations can be readily (re)assigned as sensor nodes are introduced or removed. The model includes methods for communication to accommodate these dynamic elements.

This dissertation presents a detailed description and design of a computational model that resolves these challenges using a hierarchical group mechanism. In this model, computation is tasked to logical groups and split into collective and local components that communicate hierarchically. Local computation is primarily used for data production and



*publishes* data to the collective computation. Similarly, collective computation is primarily used for data aggregation and *pushes* results back to the local computation. Finally, the model includes data-processing functions interposed between local and collective functions and are responsible for data conversion.

This dissertation also presents implementations and applications of the model. Implementations include *Kensho*, a C-based implementation of the hierarchical group model, that can be used for a variety of user applications. Another implementation, *Tables*, presents a spreadsheet-inspired view of the sensor network that takes advantage of hierarchical groups for both computation and communication. Users are able to specify both local and collective functions that execute on the sensor network via the spreadsheet interface. Applications of the model are also explored. One application, FUSN, provides a set of methods for constructing filesystem-based interfaces for sensor networks. This demonstrates the general applicability of the model as applied to sensor network programming and management interfaces. Finally, the model is applied to a novel privacy algorithm to demonstrate that the model isn't strictly limited to programming interfaces.

# Contents

<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>xvii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Application Workflow . . . . .	2
1.2 Current Programming Models . . . . .	4
1.2.1 Message Passing Models . . . . .	5
1.2.2 Restrictive Models . . . . .	6
1.2.3 Global System Models . . . . .	7
1.2.4 Limitations in Current Models . . . . .	8
1.3 A Hierarchical Group-Based Model . . . . .	10
<b>2 The Hierarchical Group Model</b>	<b>13</b>
2.1 Key Abstractions . . . . .	13
2.1.1 Logical Groups . . . . .	16

## Contents

2.1.2	Hierarchical Task Assignment . . . . .	17
2.1.3	Hierarchical Communication . . . . .	18
2.2	Kensho . . . . .	18
2.2.1	Group Implementation . . . . .	20
2.2.2	Task Implementation . . . . .	21
2.2.3	Communication Implementation . . . . .	22
2.2.4	Evaluation . . . . .	25
2.3	Discussion . . . . .	30
<b>3</b>	<b>Tables</b>	<b>32</b>
3.1	Tables Workflow . . . . .	34
3.1.1	Mapping Hierarchical Groups to Spreadsheets . . . . .	34
3.1.2	Pivot Tables . . . . .	35
3.1.3	Local Functions . . . . .	38
3.1.4	Collective Functions and Sheet Groups . . . . .	41
3.2	Implementation . . . . .	45
3.3	Evaluation . . . . .	46
3.4	Discussion . . . . .	53
<b>4</b>	<b>Applications of the Hierarchical Group Model</b>	<b>54</b>
4.1	FUSN . . . . .	55

## Contents

4.1.1	Architecture and Application Programming Interface . . . . .	56
4.1.2	Implementation . . . . .	60
4.1.3	Filesystems . . . . .	62
4.1.4	Evaluation . . . . .	66
4.1.5	Discussion . . . . .	75
4.2	Privacy Applications . . . . .	76
4.3	Negative Survey . . . . .	77
4.3.1	Selecting a Negative Category . . . . .	77
4.3.2	Reconstructing the Histogram . . . . .	78
4.3.3	Implementation and Evaluation . . . . .	80
4.3.4	Applications of the Negative Survey . . . . .	83
4.3.5	Discussion . . . . .	87
4.4	Location Anonymity . . . . .	88
4.4.1	Anonymized Locations . . . . .	89
4.4.2	Evaluation . . . . .	97
4.4.3	Discussion . . . . .	103
<b>5</b>	<b>Related Work and Conclusion</b>	<b>104</b>
5.1	Related Work . . . . .	105
5.1.1	Computational and Communication Models . . . . .	106
5.1.2	Programming Interfaces . . . . .	107

*Contents*

5.1.3	Debugging and Management Interfaces . . . . .	108
5.1.4	Privacy . . . . .	109
5.2	Future Work . . . . .	110
5.3	Conclusion . . . . .	114

<b>References</b>		<b>116</b>
-------------------	--	------------

# List of Figures

2.1	Organization of tasks in the hierarchical group model. . . . .	14
2.2	The Kensho architecture. . . . .	19
2.3	Kensho transport protocol. . . . .	20
2.4	Two local functions and a Kensho tasking function. . . . .	24
2.5	Two Kensho admission functions. . . . .	25
2.6	Photometer values over time using three different local functions. . . . .	26
2.7	Number of messages transmitted by different photometer applications. . . . .	27
2.8	Messages transmitted with respect to logical group size. . . . .	28
2.9	Mote memory usage with respect to the number of logical groups. . . . .	29
2.10	Sensor node memory usage over time. . . . .	30
3.1	An empty pivot table. . . . .	35
3.2	Pivot table compilation. . . . .	36
3.3	A simple environmental monitoring application. . . . .	37
3.4	A pivot table requesting time data. . . . .	38

*List of Figures*

3.5	A simple Tables averaging application. . . . .	40
3.6	A Tables data monitoring application with a simple filter. . . . .	41
3.7	Function compilation in Tables. . . . .	42
3.8	A Tables mobile object tracking application. . . . .	43
3.9	A screenshot of using the sheet pane to organize the sensor network into logical groups. . . . .	44
3.10	Results of the mobile object tracking application. . . . .	44
3.11	Structure of Tables pivot table replies. . . . .	47
3.12	Message transmission size with respect to the data queue size. . . . .	48
3.13	User latency with respect to the data queue size. . . . .	48
3.14	Number of messages transmitted with respect to the number of requested data. . . . .	49
3.15	Sensor node memory consumption with respect to the amount of requested data. . . . .	50
3.16	Sensor node memory consumption with respect to the number of functions stored on the node. . . . .	51
3.17	Number of transmitted messages and latency with respect to the publication window size. . . . .	52
4.1	The FUSN architecture. . . . .	56
4.2	FUSN compilation architecture. . . . .	57
4.3	FUSN pre-processing function. . . . .	58

*List of Figures*

4.4	FUSN post-processing function. . . . .	58
4.5	FUSN failure model. . . . .	61
4.6	Matlab script using FUSN to interact with a sensor node. . . . .	62
4.7	Screenshot of a user tasking and debugging a script using FUSN. . . . .	64
4.8	Memory consumption of a single task file over time. . . . .	65
4.9	Latency of opening and reading a file in FUSN with respect to file size. . . . .	67
4.10	Latency of opening and reading a file in FUSN with respect to transmission delay. . . . .	69
4.11	Latency of opening and reading a file in FUSN with respect to the number of files. . . . .	70
4.12	Latency experienced by the user for several command-line tools. . . . .	70
4.13	Number of messages transmitted with respect to file size and different compression levels. . . . .	71
4.14	Dynamic memory consumption with respect to file size. . . . .	72
4.15	Dynamic memory consumption of an executing task with respect to the size of the intermediate files. . . . .	73
4.16	File latency of reading and writing files in FUSN with respect to the number of concurrent accesses. . . . .	74
4.17	Reconstructed histograms using the negative survey. . . . .	79
4.18	Error in the reconstructed histogram with respect to the number of categories and samples. . . . .	81



*List of Figures*

4.19	Number of sensors required to maintain a particular error with respect to the number of categories. . . . .	82
4.20	Three speed distributions to characterize different traffic conditions. . . . .	85
4.21	Classification accuracy for three traffic conditions. . . . .	86
4.22	Overview of the location anonymity encoding and querying algorithm. . . . .	90
4.23	The location anonymity quadrant encoding scheme. . . . .	91
4.24	Hamming Distance properties after randomization. . . . .	93
4.25	Overview of the singleton negative database algorithm. . . . .	95
4.26	The number of matches in the location anonymity scheme with respect to $r$ . . . . .	98
4.27	The number of matches in the location anonymity scheme as the geographic distance is increased with different values of $r$ . . . . .	98
4.28	Maximum $r$ value that generated matches for different distances in the location anonymity scheme. . . . .	99
4.29	Frequency of solutions discovered by zChaff and their Hamming Distance from the original location. . . . .	101
4.30	Number of correct guesses for three strategies that attempt to obtain the original location. . . . .	102

# List of Tables

4.1	Summary of the FUSN API. . . . .	57
4.2	Functions supported by the FUSN task interpreter. . . . .	63
4.3	Static memory consumption of the tasking functions before and after compression. . . . .	65
4.4	Binary size of Mantis OS applications and different FUSN filesystems. . . . .	66
4.5	Examples of how quadrants in two extreme quadrant levels are encoded in binary. . . . .	92
4.6	Location encoding in the quadrant representation. . . . .	93
4.7	Example of a singleton negative database. . . . .	95

# Chapter 1

## Introduction

Advances in micro electromechanical systems (MEMS) sensor technology, reliable low power radios, and transistor density have fostered the creation of small-scale sensor devices. These devices form a wireless network to exchange data, coordinate their actions to sense the environment, and perform computation on environmental data. Sensors can include: thermistors, photometers, accelerometers, magnetometers, barometers, humidity detectors, microphones [82], and low-power cameras [84].

Distributed sensor networks have the potential to revolutionize disciplines that benefit from large-scale, high resolution data collection, including the natural sciences (ecosystem monitoring [70, 18], structural monitoring [48], geology [101], wildlife tracking [60]), urban scenarios (traffic monitoring [23], local weather information), and emergency scenarios (alternative communication mechanisms, radiation monitoring [16] and prediction [15]). Sensor networks provide opportunities to analyze data within the network, making data collection more efficient and powerful.

The research challenge in sensor networks is to devise software and algorithms to coordinate the exchange of relevant data between sensor nodes to produce useful information [40]. Although this challenge resembles other distributed computing paradigms, there are

several key differences. First, unlike typical distributed systems whose primary purpose is to perform some computation, the primary purpose of a sensor network is to collect data and perform limited analysis on that data. Second, node resources, including power and memory, are extremely scarce. Other resources, including radio communication and flash access, consume large amounts of power and must be used sparingly. Similarly, software must be optimized to minimize memory consumption. Finally, sensor networks are tightly coupled to the physical environment. The difficulties in programming networked resources, exacerbated by the unique sensor network traits, necessitate the use of high-level computational models to program sensor networks.

## **1.1 Application Workflow**

Programming models for sensor networks must be able to characterize the dynamic application workflow found in sensor network applications. Although a wide variety of sensor network applications have been proposed, most applications share a similar workflow and can be broadly divided into three classes: data monitoring, event analysis, and personal applications. These broad classes differ in the amount and type of data being collected, the computation performed, and the communication structure.

1. Data monitoring applications - The purpose of these applications is to collect environmental data in a particular geographic area. Such areas include remote natural settings [70, 60, 97] or man-made settings, such as cities [75, 66]. Data include radio frequency, temperature, humidity, and light levels. This data can, in turn, be used for other applications such as animal behavior modeling [14] and structural monitoring [64]. Since these applications are deployed for long periods of time, sensor nodes often filter and compress data to reduce communication and storage requirements.
2. Event analysis applications - The purpose of these applications is to collect envi-

## Chapter 1. Introduction

ronmental data and to classify events. Unlike data monitoring applications, users of these applications are less interested in the raw data and more interested in the analysis of the data. Analysis in such settings include determining traffic congestion [57, 35], classifying noise and pollution levels [85], and measuring the density of people in a particular area [1].

3. Personal applications - The purpose of these applications to collect data on the behavior of specific individuals and to communicate this data with other approved people. Such data may include the location of the person (via GPS or wireless triangulation), the health of the person (heart rate [71], diet [86]), and the online “status” of the person (such as whether they are busy, travelling, etc. [74]). This data can be used for personal or social purposes and can also be integrated into other applications. Because these applications may collect sensitive data, the data must be anonymized [56] and secured before transmission.

Although these application classes differ in many respects, they share a common computation and communication workflow. First, applications undergo a sampling phase in which the relevant data is collected. The sampling phase is often periodic, although the sampling periodicity may change over time. After sampling the relevant data, computation is performed on the data. The computation either acts on the data directly or acts on data from multiple sources. Finally, the results of the computation are communicated back to some set of nodes or to the user.

A key difference between these sensor network applications is in *when* and *where* the sampling and computation phases occur. For typical data monitoring applications, all sensor nodes share the same sampling and computation routines. These routines do not typically change over time. For event analysis and personal sensing applications, however, the sampling and computation routines may change in the context of some external event. For example, in an urban environment, sensors embedded in a collapsed building may respond by suggesting safe routes out of the building. Similarly, for personal applications,

the sampling and computation routines may change depending on the particular activity the person is engaged in. A programming model for sensor networks must accommodate this dynamic application workflow. The model must be able to control *when* and *where* the sampling and computation phases occur without unduly burdening the user.

## 1.2 Current Programming Models

The purpose of a programming model is to organize and abstract lower-level computational mechanisms. These abstractions facilitate the organization of various computational elements and simplify the construction of applications. Although any particular computational model may start at a different “level” with different assumptions about the underlying mechanisms, the lowest levels often presented to system developers are hardware registers and memory. Many of these hardware registers are used to keep track of the software execution state, such as the location of the next executable machine statement. Since there are only a limited number of registers and memory, system developers must often manually specify how different concurrent executions are specified (often in the form of an operating system process scheduler) using low-level languages such as *assembly*.

Above this layer, application developers interact with abstractions provided by the operating system. Most operating systems provide an abstraction called a *process*. Depending on the memory abstraction, processes may be referred to as *threads*. Processes abstract the notion of hardware registers and memory access. Application developers program in higher level languages (such as *C*) and interact with *variables* and *functions*. Upon creating a process, the function associated with the process runs to completion. Translating variable access and additional function invocations to lower-level memory and register accesses is done automatically by tools such as compilers and the operating system. Other popular programming models include object-oriented programming, functional programming and logic programming.

To resolve issues when moving to networked computers, single-machine computational models have been extended to include abstractions for networked communications and networked tasking. Such models range from a completely message-driven scheme in which processes are extended with methods to send and receive messages to providing a virtual, single-machine view of a networked system. Other models in this spectrum include abstracting communication in narrow ways and rigidly tasking certain computation to specific hardware. These models have been adapted for sensor networks, each with several distinct strategies and benefits.

### **1.2.1 Message Passing Models**

The most general computational model in sensor networks is the message passing model. This is the model encouraged by stand-alone operating system environments, such as TinyOS [54], SOS [52], Mantis OS [11], Contiki [34], and Maté [67]. Developers construct individual applications that execute on the sensor nodes. These applications are often constructed using a variant of the C programming language, such as NesC [46], and are often either event-based [99] or thread-based. Applications executing on the sensor nodes communicate with each other by explicitly sending messages to other nodes, and responding to specific messages. Not all implementations equally support all message types. For example, implementations using the Collection Tree Protocol [2] prevent individual sensor nodes from sending messages directly to other sensor nodes. Instead, sensor nodes must first transmit a message to the root of the routing tree, which can, in turn, transmit a message to another sensor node.

The message passing model is the most flexible of the computational models, since the model does little to restrict the types of computation that can be performed on the sensor nodes. Sensor nodes can arrange themselves in arbitrary logical topologies, and can consequently perform many types of collective operations. However, this flexibility comes at

the price of unnecessary complexity. The message passing model does not take advantage of the common application workflow to simplify programming. Consequently, constructing applications is often difficult in practice, since the developer must organize the sensor network into the relevant logical topology, manually handle all message transmissions, and coordinate the actions of all the computation occurring on the network. Finally, applications constructed using the message passing model are often fragile with respect to network changes. Messages are often sent to specific sensor nodes; if these nodes fail or disappear, the entire application may stop working. Similarly, it may be difficult to take advantage of the appearance of new sensor nodes without extensive programming.

### 1.2.2 Restrictive Models

To address these programming difficulties, other models restrict the types of computation and communication that can be performed on the sensor network. By restricting the behavior of the sensor network, these models can simplify the process of constructing applications while restricting the range of applications that can be constructed. These models may still contain explicit communication, but they often restrict the logical topologies. For example, local neighborhood systems such as Hood [102] and Abstract Regions [69] provide a set of logical topologies, such as *spanning tree* or *planar graph*, that users can choose from. Applications executing on sensor nodes, after choosing the appropriate logical topology, can refer to data residing on nearby neighbors. Communication abstractions provided by local neighborhood models target specific applications that require local neighborhood data. Other more prevalent communication patterns are not addressed by these models. Finally, these models do not abstract the type of computation performed on the network. Consequently, these models do not fully characterize application workflow.

Other models that restrict both communication and computation include tiered architectures, such as Tenet [49]. In these systems, the sensor network is explicitly defined into



discrete tiers. Each tier is responsible for a specific set of computations and communication must be between nodes of different tiers (often only between immediate tiers). Nodes within a single tier cannot communicate directly with each other. In Tenet, the lowest tier consists of the actual sensor nodes. These nodes can only execute programs constructed using a limited programming language that includes sampling and filtering routines. The nodes cannot execute arbitrary logic. In the tier above the sensors, the basestations can execute more advanced computation and can directly communicate with each other. This type of strict tiered architecture simplifies programming by constricting both communication and computation. However, the extreme inflexibility may limit implementation strategies and ultimately affect the performance of the system.

### **1.2.3 Global System Models**

Another method to address the limitations of the message-passing model is to provide a virtual, whole-system view of the network. The whole-system model often eliminates explicit communication altogether. Applications consist of a single logical program that is automatically distributed over the sensor network. When the single global application refers to distributed data, the data is automatically transferred to a designated node for computation. Examples include Kairos [51], SpatialView [78], Regiment [77] and the Declarative Sensor Network platform [26].

Although they afford many implementation strategies, these models can be difficult to implement efficiently unless the end-user interface restricts the types of computation performed by the sensor network. For this reason, Cougar [106] and TinyDB [68] impose a relational database model. Users construct SQL-like queries which are compiled and propagated to the sensor network. Responses are aggregated according to the query using pre-defined aggregation functions. Although using pre-defined aggregation functions is relatively straightforward, such systems do not provide convenient methods to construct

additional aggregation functions. Consequently, many event analysis applications are difficult to implement with these models.

### 1.2.4 Limitations in Current Models

Current computational models can be organized by whether they emphasize the sensor network as a *computational* resource or as *data* resource. Models that emphasize computation over data often ignore the workflow of existing sensor network applications. Developers are forced to manually organize their code into sampling, computation, and communication phases. Although this can result in highly optimized code (in terms of memory and network usage), the construction of more complex applications can become burdensome. Similarly, the flexibility of these tools often makes the construction of higher level end-user interfaces difficult. Finally, restrictive models that abstract communication fail to abstract the organization of computational tasks in the sensor network. Consequently, many sensor network applications remain difficult to construct using these models.

Models that emphasize data acquisition over computation, such as the relational database models and the Tenet architecture, often simplify the programming process. Simple end-user interfaces can easily be constructed for these systems, since the types of computation the user can define are strictly organized. However, such models often severely restrict the type of computation that can be defined. This, in turn, limits the scope of possible applications. For example, many event analysis applications, such as object tracking, are difficult to construct in most relational database models. Also, many of the computational and communication restrictions hinder future implementation possibilities. Applications written using such models may not be able to take advantage of future sensor nodes with better computational capabilities.

Previous models fail to adequately characterize sensor network applications partly because these models were designed for earlier, more simple deployment scenarios. These

## Chapter 1. Introduction

deployments were envisioned to consist primarily of large numbers of small, “mote” level devices [61], such as the Berkeley Mica series of devices [53]. Deployments were also to be static; once deployed, additional sensor nodes were not introduced into the system. These networks were also envisioned to be largely autonomous; these networks were often deployed in remote environments where accessibility and interaction was limited. Due to these assumptions, applications were often created and controlled by a small set of expert users. End users, including application scientists, were not expected to reprogram the sensor network.

In contrast newer deployments are expected to have a more dynamic workflow characterized by the following set of traits:

- *Heterogeneous and mobile hardware* New sensor platforms, such as cellular phones equipped with GPS and digital cameras will supplement traditional sensor platforms [5]. Many of these devices will be mobile and interact with other devices to perform a variety of tasks, such as fine-scale weather monitoring and prediction [9], traffic monitoring [93], and emergency notifications [71].
- *New environments* Sensor networks will be deployed in a wider variety of settings, including urban and personal environments such as workplaces [13], campuses, and cities [75]. In these settings, multiple parties will be able to view and share data collected by sensor networks [95], and to run multiple applications on a shared network [50]. Within a city, these sensor networks may be distributed throughout public buildings, streets, and people equipped with personal mobile devices.
- *Different types of users* Individuals will actively take part in the sensing environment [17]; people will provide and consume data, and in certain circumstances program the sensor nodes with new tasks. Not all of these people will be specifically trained to program and manage sensor networks; instead many will be regular “citizen scientists”, domain specialists, and policy managers.

## Chapter 1. Introduction

These new traits necessitate the construction of a new programming model to help users. This model has several requirements, including:

- Be able to capture the dynamic computational and communication workflow found in sensor network applications.
- Preserve the simplicity of data-driven models while preserving the flexibility of computation-driven models.
- Decouple computation and communication from specific nodes.

These requirements are not met by current programming models largely due to the inability of these models to fully characterize the workflow of sensor network applications. By characterizing the application workflow, the other goals can be more easily met.

### 1.3 A Hierarchical Group-Based Model

These requirements can be resolved by using a hierarchical, group-based mechanism (HG model). A hierarchical group model can characterize sensor network applications by focusing on the data-centric workflow of the computation, and can aid in the construction of future applications and tools. Applications constructed using this model can also exhibit relatively low message overhead.

In this model, computation is tasked to logical groups consisting of a set of sensor nodes, instead of individual nodes. Admission functions are used to determine whether a sensor node has detected a particular phenomenon and whether a sensor node should join a particular group. The admission functions are evaluated periodically, allowing group membership to change over time. Computation is manually split, by the user, into collective and local components that communicate hierarchically. Local computation is primarily used for data production, such as collecting sensor data, and *publishing* that data to

## Chapter 1. Introduction

collective functions. Likewise, collective computation is primarily used for data aggregation and analysis and *pushes* results and commands back to the local computation. Finally, the model also includes data-processing functions interposed between local and collective functions and are responsible for data conversion.

By employing logical groups, the HG model provides a simple method to organize the sensor network and assign tasks without worrying about the dynamic behavior of the environment. The model is able to accommodate both existing application patterns and future, more advanced application patterns using a simple set of computational primitives. Importantly, the HG model integrates the basestation as part of the sensor network; besides acting as conduit for the user to specify logical groups and task functions, the basestation can also perform some of the computation associated with a logical group. Finally, logical groups afford much room for different implementations. For example, the HG model does not stipulate *where* local, collective, and data-processing functions should execute.

To demonstrate the efficacy of the model, this dissertation presents two implementations, *Kensho* and *Tables* (Chapters 2 and 3). *Kensho* implements the model as a C-based library that can be integrated into current development environments. The implementation of logical groups is split between the sensor nodes and a more powerful basestation. While the sensor nodes execute local computation, collective computation is executed on the basestation. Consequently, communication occurs between the sensor nodes and the basestation.

*Tables* is an alternative implementation of the hierarchical group model that presents a spreadsheet inspired view of the sensor network. In *Tables*, users are able to construct complex data queries and specify data-driven local and collective functions. These features allow the user to create a wide range of applications using familiar graphical tools. Like *Kensho*, local functions execute on sensor nodes while collective functions execute on the basestation.

## *Chapter 1. Introduction*

These particular implementation strategies do not preclude other implementations; for example, future implementations may take advantage of heterogeneous hardware to optimize the implementation of collective functions. A discussion of alternative implementation strategies is given in Chapter 5.

To evaluate the hierarchical group model with respect to application construction, a management interface called FUSN is presented (Chapter 4). FUSN is a framework for constructing filesystem-based interfaces for sensor networks. In these filesystem interfaces, sensor nodes are presented as directories containing a set of data files. Users can then employ existing I/O libraries and tools to interact with sensor networks. Besides being generally useful, FUSN demonstrates that the hierarchical group model is flexible enough to accommodate very different workflows. In addition to FUSN, a set of novel privacy preserving algorithms that conform to the hierarchical group model are presented. Besides being useful in and of themselves, the algorithms demonstrate how very complex data-processing can be implemented using Kensho.

# Chapter 2

## The Hierarchical Group Model

The hierarchical group model is a data-centric programming model for sensor networks that includes support for flexible computation assignment and simplified communication. Unlike previous programming models, the hierarchical group model is designed specifically to accommodate sensor network application workflow and includes abstractions that are able to adapt to future hardware constraints. Using data-centric abstractions, the hierarchical group model is uniquely capable of characterizing existing application workflow while also aiding in the construction of future applications.

### 2.1 Key Abstractions

The hierarchical group model captures the sampling, computation, and communication workflow by providing a set of novel tasking abstractions. Within a distributed environment, tasking can be defined as the process of *mapping* a set of computational units onto a set of physical sensor nodes. Existing models treat tasking in one of two ways:

1. Implicitly: Systems such as TinyOS execute different code paths according to some

## Chapter 2. The Hierarchical Group Model

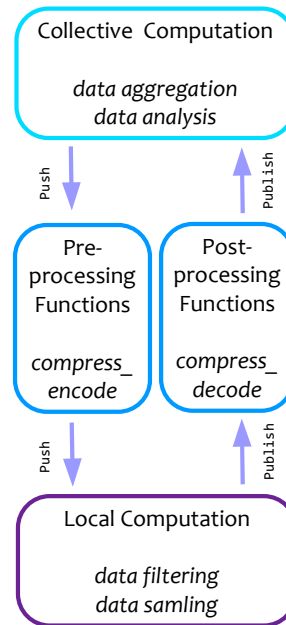


Figure 2.1: Organization of tasks in the hierarchical group model. The hierarchical group model consists of local tasks that communicate with collective tasks. Likewise, collective tasks can communicate with local tasks. The user can also specify processing functions that operate over the data.

state found on the node. These code paths, corresponding to a particular task, are hardwired in the code itself.

2. **Rigidly:** Architectures such as Tenet specify a very rigid tasking scheme. Sensor nodes execute certain types of tasks while more powerful basestations execute another set of tasks.

Both implicit and rigid architectures couple the computation to a specific set of hardware. This makes alternative implementations difficult. Similarly, such schemes make both retasking and redeployment potentially more complicated. Retasking, the process of *remapping* the set of functions according to some event, may require that some nodes execute some code paths while others do not. Manually programming several alternative code paths is difficult and error-prone. Similarly, manually programming redeployment



## Chapter 2. The Hierarchical Group Model

scenarios, in which new sensor nodes are placed in an environment, is also complicated and error-prone.

In order to properly abstract tasking and capture application workflow, the hierarchical group model decouples computation from the set of sensor nodes that execute the computation. This is accomplished using the following key abstractions:

- Event-based logical groups
- Hierarchical group task assignments
- Hierarchical communication within groups
- Data processing functions within groups

Instead of assigning a particular task to a specific sensor node, the user assigns tasks to logical groups. These logical groups are defined by external events and can include multiple sensor nodes. By dealing with logical groups, the user can organize computation around a set of dynamic events instead of individual sensor nodes. Tasks are split between local and collective tasks. Local tasks are used primarily for data production, such as sampling sensor data. After sampling, a local task can *publish* the data to collective tasks. Likewise, collective tasks are used primarily for data aggregation and analysis. These tasks can *push* results back to the local tasks. This organization abstracts the dominant computational and communication pattern found in sensor network applications. Finally, the model also includes data-processing functions that sit between local and collective tasks and are responsible for data conversion. Figure 2.1 summarizes the organization of tasks in a logical group.

Unlike other sensor network programming models, the hierarchical group model affords different implementation strategies. By using logical groups and hierarchical communication, the model allows local and collective tasks to be executed on different nodes

without affecting the definition of the user application. For example, a very simple implementation of the hierarchical group model may execute all tasks on a single, powerful basestation. Data would be periodically transmitted from the sensor network to the basestation. Local and collective tasks would then operate over this data. Hierarchical communication would consist of manipulating data locally. A purely basestation-based implementation may be useful for sensor networks with extremely limited computation.

### 2.1.1 Logical Groups

A logical group consists of a set of sensor nodes that all detect a common event. For example, all sensor nodes underneath a shadow may form a logical group. These groups, in conjunction with the functions assigned to each group, functionally divide the sensor network into different roles. These roles include simple tasks such as data collection and complex tasks, such as object tracking. Currently, many systems [42, 90] implicitly define roles; sensor nodes are defined by the code they run. Logical groups allow users to associate roles and tasks with events instead of specific sensor nodes. This abstraction simplifies the implementation of sensor network applications while sufficiently decoupling computation from sensor nodes.

In order to implement these dynamic aspects, groups are defined in the hierarchical group model using an admission function. An admission function is a binary function, unique to each group, that is executed by sensor nodes to determine group membership. The ability to access the local storage and sensor devices allows membership in a group to be defined by a wide array of data conditions. For example, a simple admission function may read one or more sensor values, perform some simple computation on the data, and return *true* if the result exceeds some threshold.

The admission function is run periodically, making logical groups dynamic. This is useful in situations when sensor nodes must perform some computation only during the oc-

## Chapter 2. The Hierarchical Group Model

currence of some specific event. For example, sensor nodes near a dangerous phenomenon (such as a large fire) may need to calculate the proximity and intensity of the fire. Similarly, sensor nodes may track the movement of objects using localized phenomena such as changes in light level, radio interference, or magnetic properties. Tasks associated with the group automatically stop running when the object moves away from the sensors.

### 2.1.2 Hierarchical Task Assignment

Task assignment is the process by which a set of functions is mapped to a set of sensor nodes. By leveraging logical groups, functions are mapped to groups instead of specific sensor nodes. This makes the system more robust to individual sensor node changes; as new nodes are introduced into the network and old nodes are removed, the Hierarchical Group model allows the system to remap the tasked functions onto alternative sensor nodes associated with the group. Similarly, tasked functions stop executing when the sensor node leaves the group.

The Hierarchical Group model provides two methods to task a group: *collective* and *local*. Locally tasked functions operate on data produced by the individual group members while collectively tasked functions operate over the data published by the group members. This organization strongly suggests that local functions execute on individual sensor nodes, while collective functions execute on one or more designated “leader” nodes. However, the Hierarchical Group model does not specify exactly which implementation strategy to pursue. These tasking abstractions are similar to the process of abstracting for-loops with a functional *map* operation, where instead of applying the same function iteratively to each data value, the function is applied to the entire list. Code that samples and reports sensor data can be locally tasked to the appropriate group without specifying individual sensor nodes. Similarly, instead of assigning code that aggregates data to run on a pre-defined leader node, the user can specify that this code be tasked as a collective

function.

### 2.1.3 Hierarchical Communication

By taking advantage of the hierarchical tasking scheme, the Hierarchical Group model abstracts common communication patterns by ensuring that data is accessed and communicated strictly according to the task structure. Locally tasked functions communicate with collective functions using a *publish* command. Collective functions, in turn, communicate with local functions using a *push* command. Like other tiered architectures, local functions on one sensor node cannot communicate directly with other local functions; instead, these functions must use a combination of *publish* and *push*. These operations abstract many-to-one and one-to-many communication patterns [25], and relieve the application developer from dealing with complex communication protocols. Like hierarchical task assignment, the HG model does not specify how hierarchical communication is to be implemented. So long as applications are written within the context of this model and the API associated with a particular implementation, the application will operate over different implementation strategies.

## 2.2 Kensho

Kensho is a C-based implementation of the hierarchical group model developed primarily to validate and quantify the communication overhead associated with the model. Instead of executing both local and collective tasks on the basestation, Kensho implements the hierarchical group model by assigning local computation to the sensor network while assigning collective computation to more powerful basestations (Figure 2.2). This tiered implementation ensures that both local and collective computation can be used to reduce the overall number of message transmissions. However, this implementation also requires

## Chapter 2. The Hierarchical Group Model

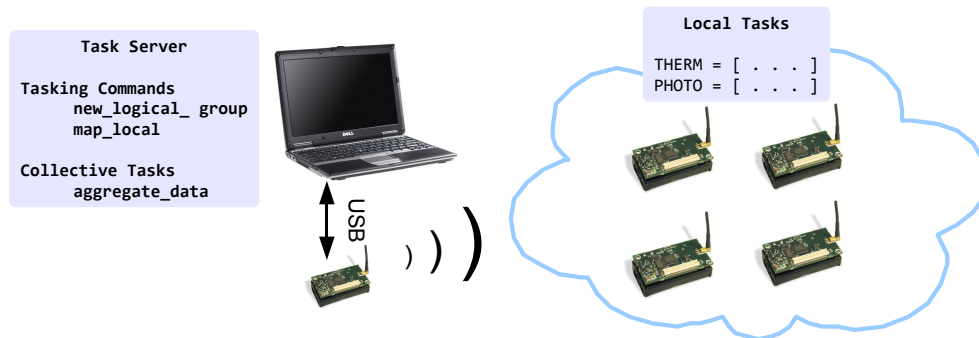


Figure 2.2: The Kensho architecture. Locally tasked functions execute on the sensor nodes, while collectively tasked functions execute on the basestation. The user interacts through the basestation which accepts and executes tasking instructions.

more complex group management protocols (since the sensor nodes must maintain their own state) along with more complex message protocols.

In order to provide this tiered implementation, Kensho consists of three major components: a tasking runtime responsible for specifying groups and mapping computation that executes on the basestation, a collective function interpreter executing on the basestation, and a local function interpreter executing on individual sensor nodes. The local function interpreter, besides evaluating the local functions, also manages the group state of the sensors. Each set of components is implemented as a C-library and provides a C-based API.

Once the user has programmed the necessary tasking commands on the basestation, the tasking program is compiled like a typical executable. Once executed, the tasking program transmits the tasking commands to the sensor network. The implementation on the sensor nodes is built using Mantis OS [11], a multi-threaded, preemptive operating system for common sensor node platforms. Since Mantis does not currently support dynamic linking, all function definitions, such as local tasks and admission functions, must be defined during the static linking stage. The tasking server components are implemented as normal C-libraries that can be linked to form an executable.

## Chapter 2. The Hierarchical Group Model

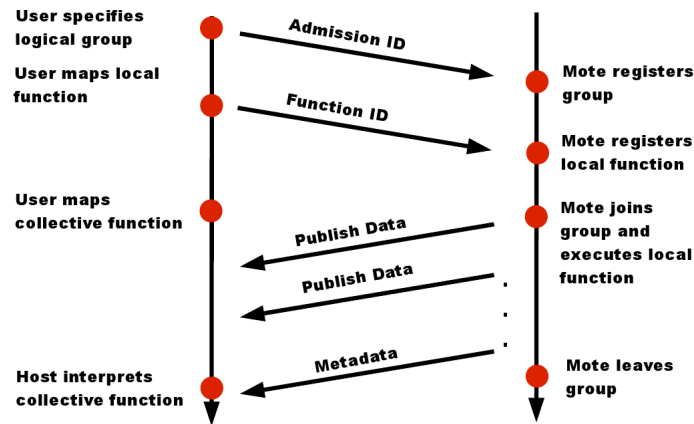


Figure 2.3: The Kensho transport protocol. Commands are executed on the basestation and transmitted to the sensor nodes. Sensor nodes, in turn, transmit publication data back to the basestation.

Since most tasking servers do not come equipped with the proper radio to communicate with the sensor nodes, communication between the basestation and sensor network is performed using a mote “bridge”. The bridge is simply a sensor node is attached to a basestation via USB. This node is responsible for relaying commands and data to and from the sensor network. For wireless, tree-based routing protocols, the bridge also serves as the root of the routing tree.

### 2.2.1 Group Implementation

In Kensho, logical groups are created using a C-based API. The primary function used is the “new\_logical\_group” function. This function accepts an admission function identifier and the suggested sampling rate for the admission function. The only requirement of admission functions is that they are binary (can only return *true* or *false*). Since admission functions are evaluated periodically, Kensho also provides a way to pass a “scratchpad” to admission functions to maintain state. Currently, admission functions are identified by searching through a designated file on the sensor nodes. These files contain the function

## *Chapter 2. The Hierarchical Group Model*

definitions along with the unique function identifiers. In the future, these files will be generated automatically.

Since the admission function must execute on the sensor nodes, the definition of the admission function must be statically compiled and located on the sensor node. Once the logical group is specified (and the tasking program is compiled and executed), the new logical group command is propagated to all the nodes in the sensor network. Since any node may join the logical group some time in the future, each node registers the potential group and starts a thread to periodically evaluate the admission function.

When the admission function determines that the node should join the logical group, the sensor node registers the group internally as “active”, searches for any local tasks that were assigned to the group, and finally starts a thread for each of the assigned local tasks. If, in the future, the sensor node leaves the group (the admission function returns “false”), the group internally removes itself from the “active” list. Once activated, local task threads are not completely stopped. Instead, the threads are paused and removed from the execution queue so that they are not scheduled until the node rejoins the group. This ensures that nodes that temporarily leave the group do not need to reinitialize all the local tasks. In the future, once the sensor node has been removed from a group for a long period of time, local tasks may be completely stopped.

### **2.2.2 Task Implementation**

Kensho provides two separate API calls for collective and local tasking, “map\_local” and “map\_collective”. Both functions accept a unique function identifier and a unique group identifier to which the function is assigned. These group identifiers are generated after specifying a logical group. Like the admission functions, these functions must be statically defined and compiled for the sensor nodes. Once the tasking program is compiled and executed, local task commands are propagated to the entire sensor network. Upon

## *Chapter 2. The Hierarchical Group Model*

receiving a local task command, each sensor node must store and register the function. A sensor node, upon joining the appropriate group, then creates a thread to execute the assigned local tasks.

Collective tasks, unlike local tasks, do not execute on the sensor nodes. Instead, collective tasks reside and execute on the basestation. This particular implementation has several benefits. First, the strategy relieves the sensor network from dynamically choosing the sensor node that will execute the collective task. Distributed election algorithms can be complex and often involves complicated communication tradeoffs [10]. This strategy also simplifies the routing strategy since a single routing tree rooted at the basestation can be used for the entire sensor network. Finally, since many collective tasks may require complex aggregation from multiple sensor nodes, executing the task on the basestation may reduce the overall computation latency.

### **2.2.3 Communication Implementation**

In Kensho, communication is accomplished by referring to named elements. This simplifies the API while still conforming to the hierarchical group model. The communication abstractions are provided using the “publish\_data”, “push\_data”, and “collect\_data” function calls. These functions accept the name of the data along with the group identifier. Since local tasks are executed on the sensor node while collective tasks are executed on the basestation, “publish\_data” transmits the data from the sensor network to the basestation. Similarly, “push\_data” transmits data from the basestation to the relevant sensor nodes. Sensor nodes, upon receiving data, determines if they belong in the correct group and then store the data appropriately.

Kensho currently relies on an underlying routing protocol to transport packets throughout the sensor network. Although Kensho does not rely on the semantics of any specific routing protocol, the protocol must provide the ability to route packets from a basestation



## Chapter 2. The Hierarchical Group Model

to all nodes in the network, along with the ability to route packets from a sensor node back to the basestation. The collection tree protocol [2, 44] (CTP) provides these features, along with the ability to send messages to specific nodes, using a multi-root tree-based scheme. CTP also provides a port-based demultiplexing scheme. Other routing schemes, including MintRoute [105] also provide similar services and could have been used just as easily. For the current implementation, Kensho employs a single-root routing tree and a single port. The Kensho tasking server and the associated mote bridge serve as the root of the tree.

Currently, the transport protocol used between the sensor network and tasking server uniquely tags each message as a tuple  $\langle grp_{id}, session_{id}, origin \rangle$ . Overall, the Kensho packet header is a total of 9 bytes with a maximum data payload of 40 bytes. In order to minimize the number of transmissions due to the relatively low data payload, each data vector is aggregated into as many data packets as possible. After sending all the relevant data, the sender then transmits a *metadata* packet indicating the number of packets that were transmitted, and the total number of bytes contained in the message. This is then used to reconstruct the original data vector. Figure 2.3 illustrates the flow of messages from the basestation to the sensor nodes.

In the case that a user application transmits a data object less than 40 bytes, only a single transmission is made without any metadata packets. This is more common for messages sent by the basestation to the sensor nodes that usually consist of short commands. Currently the transport protocol does not include any reliability or flow-control mechanisms. Each node, upon issuing a “publish\_data” command, immediately sends the packet to CTP which queues and sends the packet after some random delay to avoid intra-path interference. However, this scheme is insufficient and does not avoid queue overflow or intra-path interference. Applications using Kensho must currently provide their own intra-path interference prevention mechanism.

Unlike the other two function calls, “collect\_data” is a blocking call used to retrieve local data; collective tasks retrieve data resident on the basestation, while local tasks re-

## Chapter 2. The Hierarchical Group Model

```
int main(int argc, char** argv)
{
    ...

    group = new_logical_group(ADMISSION_FN_ID, 256);
    map_local(group, LOCAL_FN_SENSE);
    map_local(group, LOCAL_FN_COLLECT);
    map_collective(group, COLLECTIVE_FN_COLLECT);
}

void local_fn_sense(void)
{
    ...

    for(;;) {
        dev_read(DEV_MSP_TSR, &photo, 2);
        add_data(store, "PH", sizeof(uint16_t), &photo, NUMBER_TYPE);

        mos_thread_sleep(SAMPLE_INTERVAL);
    }
}

void local_fn_collect(void)
{
    ...

    for(i = 0; i < 30; ++i) {
        all_data = collect_data(store, "PH", 1, last_time, TIMEOUT);

        container = (struct data_container*)all_data->head->data;
        last_time = container->time;

        memcpy(&data.data[data.size], &container->time, sizeof(uint16_t));
        data.size += sizeof(uint16_t);

        memcpy(&data.data[data.size], &container->data.i, sizeof(uint16_t));
        data.size += sizeof(uint16_t);
    }

    publish_data(group_id, "ph", &data);
}
```

Figure 2.4: Two local functions and a Kensho tasking function. The local functions sample the sensors, stores the results, and eventually publishes the data. The tasking function specifies the logical group and maps functions to the group.

trieve data resident on the sensor node. If the user requests data that doesn't exist, the calling thread is blocked until the data is locally inserted (by another local process or by the other communication mechanisms).

## Chapter 2. The Hierarchical Group Model

```
uint8_t admission_fn_filter(void* arg)
{
    ...

    dev_read(DEV_MSP_TSR, &photo, 2);

    if(photo > 100)
        return TRUE;

    return FALSE;
}

uint8_t admission_fn_window(void* arg)
{
    ...

    if(arg == NULL) {
        last_time = 0;
        arg = malloc(sizeof(uint16_t));
    }
    else memcpy(&last_time, arg, sizeof(uint16_t));

    avg = 0;
    group_id = my_group();

    all_data = collect_data(store, "PH", 3, last_time, TIMEOUT);

    for(temp = all_data->head; temp != NULL; temp = temp->next) {
        container = (struct data_container*)all_data->head->data;
        avg += container->data.i;
    }

    memcpy(arg, &container->time, sizeof(uint16_t));
    avg /= all_data->size;

    if(avg > 100)
        return TRUE;

    return FALSE;
}
```

Figure 2.5: Two Kensho admission functions. The first admission function performs a simple threshold-based filtering, while the second function calculates the average sensor value before filtering.

### 2.2.4 Evaluation

In order to evaluate the current Kensho implementation and the hierarchical group model, several applications using the described API were developed. These applications include a simple data monitoring application that collects sensor data (photometer, thermistor, and humidity) and transmits these data to a basestation on a periodic basis. This application was then extended to include both simple and advanced filtering mechanisms.

Figure 2.4 illustrates three separate functions: the main function that creates the sensor network group and assigns the local and collective tasks and two locally tasked functions.

## Chapter 2. The Hierarchical Group Model

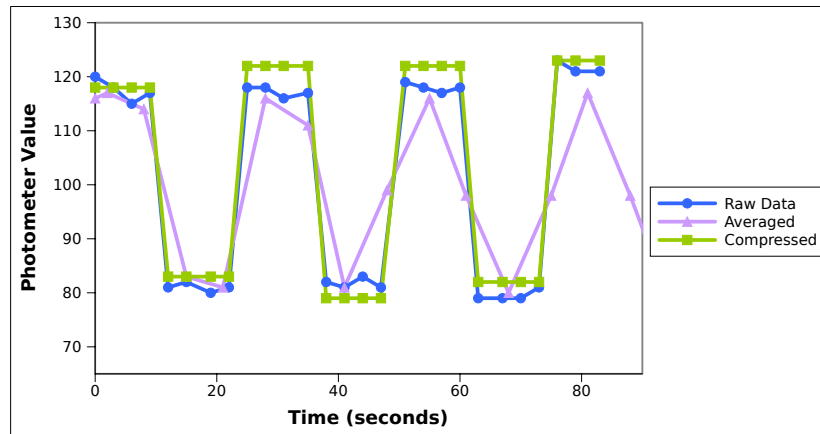


Figure 2.6: Photometer values over time using three different local functions. The raw strategy records and publishes photometer values without any processing. The average strategy averages two photometer values and publishes this value. The compressed strategy only records and publishes changes in slope.

The main function is relatively simple compared to the functions that actually perform work, and can be defined independently of the other functions. This demonstrates how Kensho and the Hierarchical Group model separate the tasking process from the function definitions. The first locally tasked function simply reads photometer data using the Mantis OS device interface and stores the data in a globally shared datastructure. The second local task then reads the data using the “collect\_data” function call and eventually publishes the data. Although the application also tasks a collective function, the collective function simply reads the published values and is omitted.

Figure 2.5 illustrates two different admission functions. The first is a simple filtering function that reads a sensor value and returns *true* only when that value exceeds some threshold. The second filtering function is more complex and averages the 3 most previous values and checks the average against some threshold value. Used in combination with the main function and tasks, these function definitions constitute a complete Kensho application.

In Kensho, most applications consist of a set of local tasks (that collect and filter data),

## Chapter 2. The Hierarchical Group Model

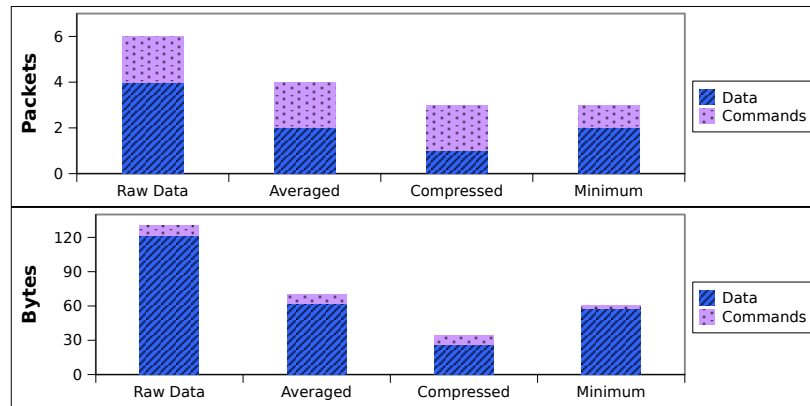


Figure 2.7: Number of messages transmitted by different photometer applications. The raw data implementation transmits the most data, while the compressed implementation transmits the least. Compared to a minimum message model that assumes the transmission of raw data but with little overhead, the compressed implementation uses the same number of packet transmissions.

a publication phase (that transmits the data to a basestation), and a collective task (that retrieves the published values and analyzes them). As such, the primary way to reduce the number of transmissions is to use local functions to compress and filter data before transmission. For example, Figure 2.6 illustrates the results of the three previously mentioned applications. The first application fetches and publishes raw photometer data over time. The second application averages 2 photometer values and publishes this average. Finally, the third application keeps track of the changes in photometer values and publishes these changes.

As Figure 2.7 illustrates, employing simple filtering mechanisms can drastically reduce the number of messages transmitted without reducing the ability to reconstruct the original data. In order to illustrate the amount of overhead associated with transmitting the raw photometer values, the applications were also compared to the number of transmissions it would take to transmit the raw values assuming minimal overhead. Even compared to this lower bound, employing local filtering functions still reduces the number of transmissions. By providing a straightforward way to construct and map local functions, Kensho

## Chapter 2. The Hierarchical Group Model

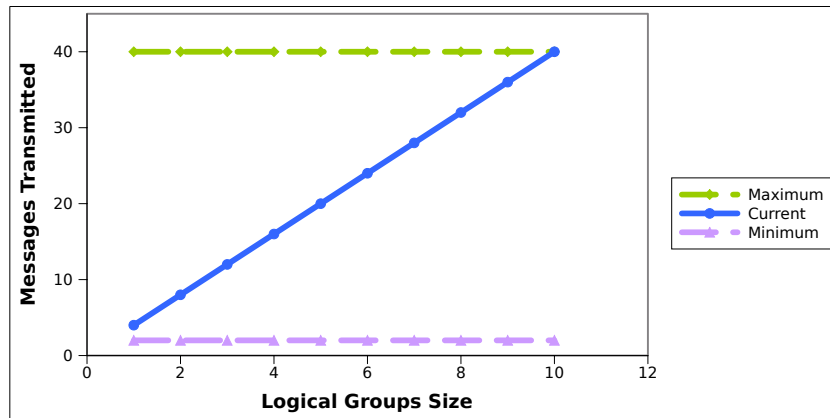


Figure 2.8: Messages transmitted with respect to logical group size. Only active members of logical groups publish messages to the basestation. As such, the number of messages scale linearly with respect to logical group size.

application developers can easily switch between different local functions.

Besides using local functions to reduce message transmission, users can also assign sensor nodes to logical groups. By doing so, only the activated sensor nodes transmit data back to the basestation instead of all sensor nodes. As such, even if local tasks publish all their sensor data, the number of transmitted messages scales with the size of logical groups as opposed to scaling with the size of the entire sensor network (Figure 2.8). In the future, alternative collective task implementations may reduce these values even further.

In Kensho, each sensor node can be a member of multiple logical groups. However, the number of logical groups is constrained by the static memory for the admission functions, the dynamic memory used for group membership data, and the memory used for stack space for the admission function thread. In order to evaluate the different memory use, the number of logical group commands the sensor node received was varied from 1 to 10. Although the sensor node registered the logical group, the group was not activated.

As Figure 2.9 indicates, the static space used to store the admission functions consumes the most memory. Although the definition size of these functions varies according to the

## Chapter 2. The Hierarchical Group Model

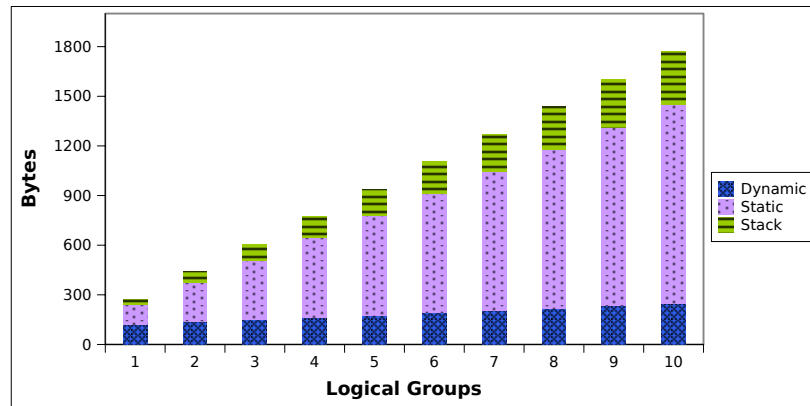


Figure 2.9: Mote memory usage with respect to the number of logical groups on a single sensor node. Overall, the stack space allocated for threads grows faster than the dynamic memory used by the functions. The static function size also consumes much space, although that space is allocated from the program memory space.

complexity of the function, 120 bytes was chosen as typical (the size of the averaging admission function shown earlier). Although the static size constitutes the largest use of space, most sensor platforms have a relatively large amount of program space separate from the data memory (48 kb for the TelosB). Besides the static space consumed by these functions, the stack space allocated for each function thread also constitutes a major source of data usage. Currently each function thread is statically initialized with a 128 byte stack.

In order to understand the memory use of a sensor node over time, dynamic memory use and stack allocations were also measured during several important Kensho operations. A new logical group was created and locally tasked with a single function. The sensor node then joined the group and began executing the local task. As Figure 2.10 illustrates, the sensor node allocates a relatively large amount of memory during the initialization stages. It is during these stages that datastructures to hold local values (accessed by “collect\_data”) are allocated. Subsequently approximately 40 bytes are allocated for group and function storage. As for stack space, Kensho initializes a thread for handling communication and commands. Subsequently, a thread is also initialized to evaluate the admission function, and eventually a thread is initialized for executing the local function.

## Chapter 2. The Hierarchical Group Model

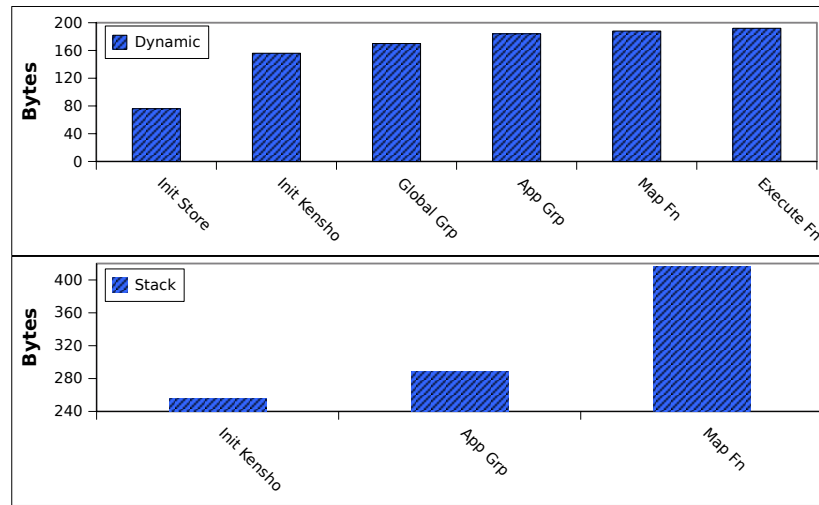


Figure 2.10: Sensor node memory usage over time. After dynamically allocating memory for local data storage, dynamic memory allocation increases slowly for the other Kensho operations. Most of the memory consumption comes from the allocation of stack space to evaluate tasked functions.

### 2.3 Discussion

This chapter introduced the hierarchical group model, a programming model designed to characterize sensor network applications and aid in their construction. The hierarchical group model abstracts distributed tasking by using event-based logical groups. These logical groups allow users to construct applications around dynamic events instead of specific sensor nodes. Within these groups, sensor nodes can be tasked either locally or collectively. Local and collective tasks allow users to abstract common computational patterns, such as data sampling and aggregation. Finally, the hierarchical group model provides communication between local and collective tasks, abstracting the dominant communication pattern found in sensor network applications.

This chapter also introduced Kensho, a C-based implementation of the hierarchical group model. Kensho builds on a preemptive, threaded operating system and treats tasks as threads. Local tasks execute on the sensor nodes, while collective tasks execute on the



## *Chapter 2. The Hierarchical Group Model*

basestation. Local and collective tasks can then communicate using a hierarchical communications API. Since the communication is between the sensor node and a basestation, existing routing schemes (including tree-based schemes) can be used.

Although Kensho exposes a C-based interface, the hierarchical group model does not strictly require the use of any particular language. It also does not require that the implementation uses threads as the primary unit of computation. Although threads offer a convenient abstraction for most computation, they are not necessarily optimal for every application. For example, applications that respond to common events may benefit from an event-based computation model. The hierarchical group model can be used in event-based systems by tasking local event handlers and collective event handlers. Logical groups and hierarchical communication would continue to play the same roles.

The next chapter examines an alternative implementation of the hierarchical group model that employs data-driven functions as the underlying computational unit. These functions, unlike threads, are automatically executed whenever data the function refers to is updated. This is similar to event-driven computation, but where the events are defined implicitly within the function. This implementation presents these functions within the context of a spreadsheet programming environment for sensor networks. This serves to demonstrate the flexibility of the hierarchical group model and how it can be applied to different programming environments.

# Chapter 3

## Tables

As sensor network deployments encompass a wider array of environments and users, new end-user interfaces will need to emphasize ease-of-use. Historically, creating, deploying, and managing a sensor network application consisted of either hiring a set of experts or using a pre-existing set of software. Hiring experts is potentially the most flexible but also the most expensive strategy. This particular strategy is also unscalable, since the number of potential applications may exceed the number of people qualified to construct and manage such applications. Using pre-existing software is much more scalable, but not very flexible. Adapting existing software for a new application may be difficult or in some cases impossible.

In order to resolve these issues, end-user interfaces must be designed to allow users to easily create and manage sensor network applications on their own. This can be accomplished by adapting existing software concepts, such as filesystems and spreadsheets, and applying them to sensor networks. By adapting these concepts, users will be able to transfer their existing knowledge base and skills. The challenge associated with adapting pre-existing interfaces for programming sensor networks is to limit arbitrary interaction (otherwise the interface becomes too complicated) while keeping the interface flexible

### *Chapter 3. Tables*

enough to create many types of applications.

Prior work in this area, including the author's initial work on a spreadsheet programming interface [55] and subsequent work by Woo et al. [104], demonstrates the challenges of this approach. For example, the work presented by Woo et al. lacks an integrated programming model, thus limiting their interface to simple data collection. Their system performs the computation centrally on the spreadsheet and does not provide a mechanism to program the sensor nodes directly. Fortunately, the hierarchical group model provides a foundation from which to adapt a spreadsheet interface for directly programming sensor networks.

Tables is a spreadsheet inspired end-user interface that employs the hierarchical group model to program sensor networks. By emphasizing a spreadsheet interface, Tables minimizes the number of new concepts needed to program a sensor network. Spreadsheets are familiar to many computer users and include advanced data manipulation functionality. In a typical spreadsheet environment, such as Microsoft Excel, users are presented with data placed along multiple axes (rows, columns, and sheets). The data can then be manipulated by functions that operate over a range of cells. The output of the function can, in turn, be consumed by additional functions. More recently, advanced spreadsheet applications also include functionality (pivot tables) to automatically organize data according to user-specified parameters.

By exporting a spreadsheet environment and employing the the hierarchical group model, Tables meets the following goals:

- Allow application specialists and casual users to easily create simple programs.
- Allow advanced users to create complex applications using the same set of constructs.
- Minimize the difficulty in learning the environment by re-using familiar concepts

and interfaces.

## 3.1 Tables Workflow

Tables, by adopting the spreadsheet metaphor, emphasizes an interactive, iterative method of programming. Data is collected from the sensor network and organized using a graphical tool called the pivot table. Once the data is viewed, the user has the option of inputting functions that operate over that data. This function, in turn, is propagated to the sensor network to either generate new data or filter existing data. Finally, the user can create a new pivot table to view the updated data. This workflow encourages users to treat data viewing as an integral part of the programming process. Similarly, this workflow is very forgiving; during any step in this process, users are left with a functioning application.

### 3.1.1 Mapping Hierarchical Groups to Spreadsheets

Tables, like Kensho, uses the hierarchical group model to simplify sensor network programming. However, instead of using threads as the underlying computational unit, Tables uses spreadsheet-like functions that operate over sensor data. Like spreadsheet functions, functions in Tables are automatically executed whenever dependent data changes. This encourages a simple function-oriented programming method where users don't need to explicitly define control flow. Like Kensho, Tables also employs logical groups. However, logical groups are not defined explicitly in Tables. Instead, logical groups are constructed implicitly by requesting certain data and specifying functions over that data. These differences make transitioning from a thread-based model more difficult, but demonstrates the general applicability of the hierarchical group model.

Finally, Tables also includes the ability to specify both local and collective functions. In accordance with the hierarchical group model, local functions communicate with col-

## Chapter 3. Tables

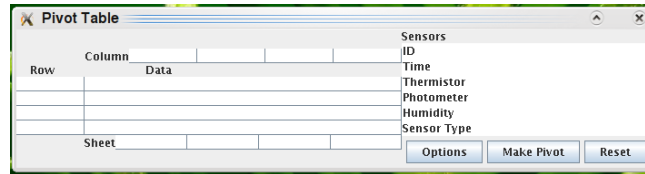


Figure 3.1: An empty pivot table. The user creates the pivot table by clicking and dragging an item name from the sensor list to one of the four panes. The data pane specifies what data to view, while the other panes specify how to organize the data. The user can also optionally specify a recurrence.

lective functions using a “publish” mechanism. Unlike Kensho, however, communication in Tables is implicit. The user does not explicitly define communication mechanisms. Instead, by referring to specific data required by the function, the data is automatically transmitted from the sensor nodes.

### 3.1.2 Pivot Tables

One of the key elements used in Tables is the pivot table. The pivot table provides a miniature representation of the spreadsheet by which to construct and organize data queries (Figure 3.1). This is an important element in Tables applications since this is the only way to view data produced by the sensor network. Additionally, advanced features (such as collective functions) can only be specified after using a pivot table.

The right hand side of the pivot table contains a list of data items that the user can query. Users click and drag these data items to one of several panes to construct a query. The panes are organized into one data pane and three metadata panes. These metadata panes represent the standard spreadsheet axes (row, column, and sheet). The user specifies which data to view by dragging a data item onto the data pane. Dragging an item onto one of the metadata panes specifies how the items in the data pane are to be organized in the spreadsheet. Each of these panes, with the exception of the sheet pane, can contain

### Chapter 3. Tables

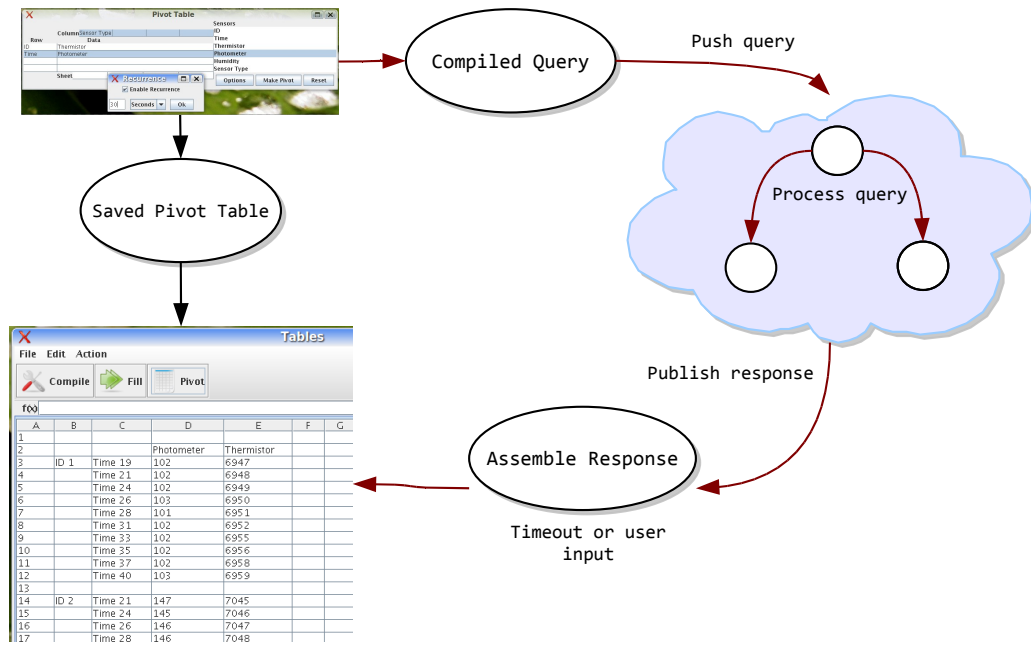


Figure 3.2: Pivot table compilation. The pivot table is first compiled and then transmitted to the sensor network. Each sensor node executes a query processor that accepts the pivot table and forms a response. The response is then transmitted back to the basestation.

multiple items.

All Tables applications start in a default state where each sensor node is preloaded with the Tables runtime. By default, the Tables runtime periodically samples photometer, thermistor, and humidity data. The default sampling rate is set to 1 second, although the value can be changed. Each sensor data is stored separately in a circular buffer of some known size. These sensor data items are also automatically listed in the pivot table data items list. The pivot table data items list also contains additional sensor metadata such as node ID and the available sensor types. As the user interacts with Tables and creates functions that assign new data, the pivot table list will be automatically updated.

Once the user specifies a pivot table, the pivot table is compiled and propagated onto the sensor network (Figure 3.2). Sensor nodes, upon receiving a pivot table request, will

## Chapter 3. Tables

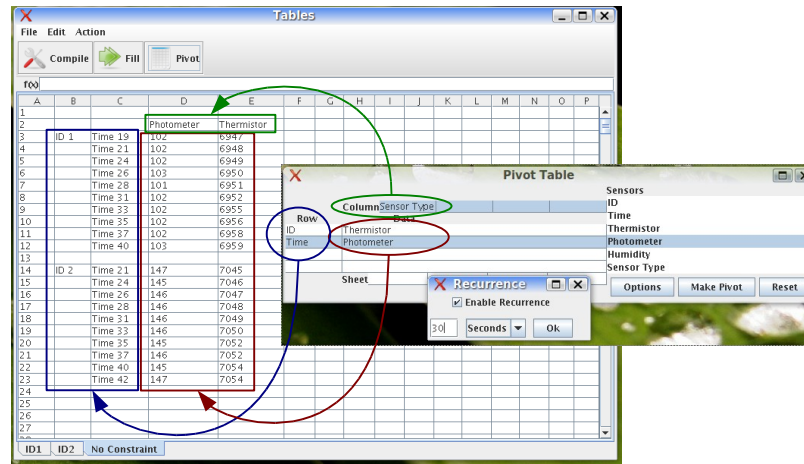


Figure 3.3: A simple environmental monitoring application. The user has requested to view thermistor and photometer data, organized by the node ID, time, and the sensor type. The layout of the response is governed by the specification of the pivot table. The pivot table and associated response are updated according to the recurrence time.

construct an appropriate response and transmit the response back to the basestation. The response consists of the entire queue of requested data along with the requested metadata. Because the data queue may be large, the response is often split up into multiple packets. After assembling all the responses from the sensor network, Tables organizes the responses to a final view according to the original pivot table specification. Like a normal spreadsheet environment, Tables lays the data out along a set of two dimensional tables (a *sheet* in spreadsheet parlance).

Although the pivot table appears to be a simple tool, it can still be useful by itself. The user can create a simple environmental monitoring application by simply dragging the desired sensor data onto the data pane. Additionally the user can organize the sensor data by node ID, time, and the name of the sensor by dragging these items on the metadata panes. If the user wants to view the data periodically, the user can specify a recurrence time. Doing so will result in a complete application as illustrated in Figure 3.3.

The pivot table is not limited to viewing sensor data. By clicking-and-dragging the *time*

## Chapter 3. Tables

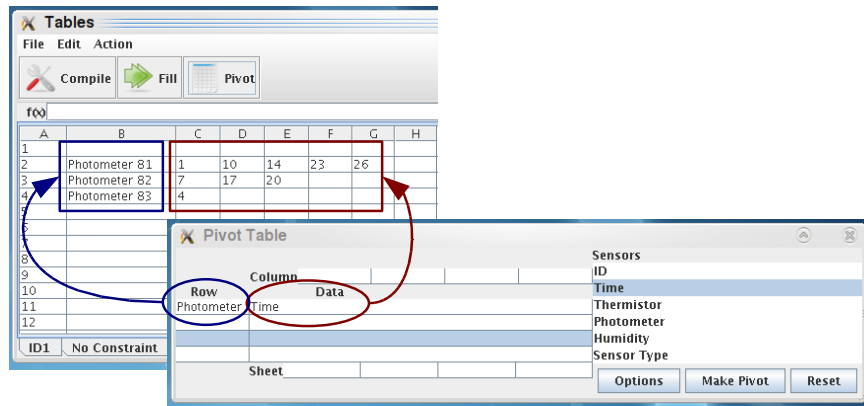


Figure 3.4: A pivot table requesting time data. The user has requested to view all times associated with a particular photometer value. Every time index a particular photometer value appears is displayed to the user. This can be used to determine when and how often particular events occur.

value onto the data pane and the *photometer* value onto one of the metadata panes, the user can request to view all times associated with a particular photometer value (Figure 3.4). Subsequently, the time indexes when a particular photometer value appears are displayed to the user. This can be used to determine when and how often particular events occur.

### 3.1.3 Local Functions

Tables, like more traditional programming environments, also allows users to construct functions that operate over sensor data. These functions, in conjunction with pivot tables, can be used to construct full-fledged applications. Like typical spreadsheet functions, functions in Tables are data driven. The functions are automatically evaluated whenever dependent data is updated. This, in turn, may generate additional data values that trigger the evaluation of other functions.

Tables provides arithmetic functions, boolean operators, and vector functions. All functions can operate over sensor data by simply referring to the name of the sensor device.



### Chapter 3. Tables

For example, the string “Photometer” refers to the latest photometer value. Similarly, functions can refer to other data items stored on the sensor node.

Vector functions, such as *average*, *sum*, and *min*, take a window size and the name of the data to operate over. For example, users can specify that the *sum* function operates over the last three thermistor values using the syntax “sum(3, Thermistor)”. This is not ideal, however, since most spreadsheet users are accustomed to specifying the range using syntax similar to: *A5:A10*. However this extension is not necessarily crucial and remains a subject of future work.

In addition, Tables provides conditional functions that allow users to take different actions depending on the results of other computation. Finally, Tables provides assignment functions that generate new data. Upon evaluating an assignment function, Tables will store the new value on the sensor node and update the pivot table list with the assigned name. These values can also be used in other functions by referring to the assigned name.

The Tables programming model supports two types of functions: local and collective. Local functions operate directly on the data produced by the sensor node. For example, local functions may consist of data compression and filtering. Consequently, these functions execute on the sensor nodes. Collective functions, on the other hand, operate over data from *multiple* sensor nodes. Consequently, collective functions execute on the base-station. Examples of these functions include aggregation and data classification.

Users create new functions by simply typing into an empty cell. Functions are designated local or collective depending on the sheet in which they are located. Like a standard spreadsheet, each sheet in Tables has a name. By default, the Tables interface provides a set of sheets with an exhaustive list of all node IDs. If the function is typed in a sheet containing the name “Node = *n*”, where *n* is some node ID, the function is designated local. Otherwise, if the sheet has a different name, the function is designated collective.

Once the user has created a local function, the function is compiled and transmitted

### Chapter 3. Tables

```
Local Functions:  
  (1) =AVG := average(2, Photometer)  
  
Collective Functions: (None)  
Pivot Tables:  
  (2) Data = AVG : metadata = Node ID, Time, Sensor Type
```

Figure 3.5: A simple Tables averaging application. This application consists of a single average function. The function is automatically evaluated whenever the photometer data is updated. Afterwards the user can construct a pivot table to view the average photometer data.

to the relevant node. The Tables interface provides a convenient menu item (“Fill”) that allows users to quickly replicate the function across multiple sensor nodes. Once a sensor node receives a function from the basestation, the sensor node stores and examines the function for dependent data. For example the function “if(Photometer > 50) ...” depends on the latest photometer data. This function is automatically evaluated whenever the photometer data is updated.

Local functions can supplement simple monitoring applications by compressing and filtering data on the sensor node. This allows the user to reduce the number of messages transmitted back to the user. A simple application that stores an average of sensor node is show in Figure 3.5. This averaged data is stored on the sensor node until a pivot table requests the data.

Another application is illustrated in Figure 3.6. This application is more complex since the user specifies a function that records changes in the photometer readings. The user begins by first assigning the “PREV” variable. This causes the PREV variable to be stored on the sensor node and listed in the pivot table. The user then specifies a function utilizing a conditional-statement that tests for differences in the current and previous photometer values. When the value exceeds a particular threshold, the previous value is updated and stored on the sensor node. By requesting the previous values using a pivot table, the user can reconstruct the photometer values.

## Chapter 3. Tables

```
Local Functions:  
(1)  
=PREV := 0  
=if( |PREV - Photometer| > 10) PREV := Photometer  
  
Collective Functions: (None)  
Pivot Tables:  
(2) Data = SLOPE : metadata = Node ID, Time, Sensor Type
```

Figure 3.6: A Tables data monitoring application with a simple filter. The user first specifies basic filtering functions that record changes in the data. These data changes are automatically stored on the sensor node. After letting these functions run for a time, the user can specify a pivot table to retrieve these data changes.

### 3.1.4 Collective Functions and Sheet Groups

Tables also provides a method to construct *collective* functions. Collective functions, unlike local functions, operate over a logical group of sensor nodes. In order to create a collective function, the user types in the desired function in a sheet representing the desired sensor nodes. For example, a sheet with the name “Node ID = 5” creates local functions. However, a function in a sheet with the name “Sensor =  $\tau$ ” creates a collective function since the sheet doesn’t identify a specific sensor node.

Sensor nodes where “Sensor” is equal to  $\tau$  form a logical group that evaluates associated collective functions. These constraints are formed using a pivot table. The user simply clicks and drags the “Sensor” item onto the sheet pane. After evaluating the pivot table, the Tables interface is populated with a set of sheets with differing “Sensor” values.

After typing in the function, the constraint “Sensor =  $\tau$ ” is transmitted to the sensor network. Sensor nodes, upon receiving the constraint, store and evaluate the constraint. This evaluation occurs whenever the constraint data (“Sensor”) is updated. If the sensor node matches the constraint, the sensor node joins the logical group representing all nodes that participate in the collective operation. Since the constraint is evaluated periodically, sensor nodes are able to leave and join the logical group dynamically. Figure 3.7 summa-

### Chapter 3. Tables

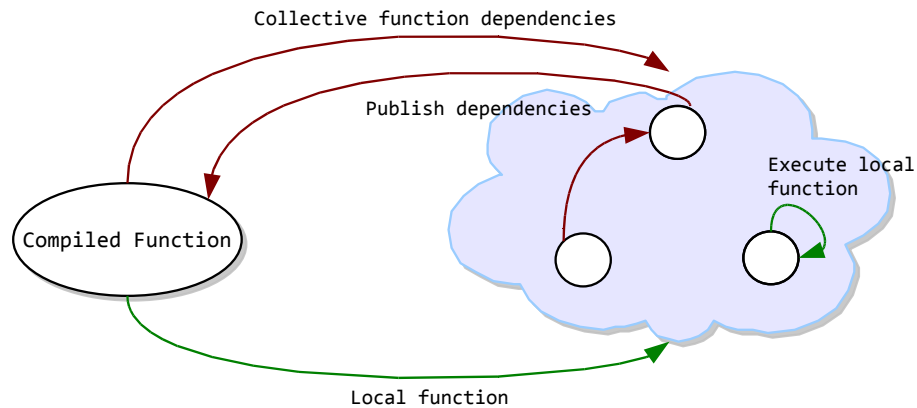


Figure 3.7: Function compilation in Tables. After users specify a function, the function is propagated to the sensor nodes. For local functions, the entire compiled function is transmitted. For collective functions, only a description of the dependent data is transmitted.

rizes these events.

Since collective functions operate over logical groups, collective functions require data from group members to be transmitted to the basestation. To accomplish this, the basestation associates a publication request with the logical group constraint. These publication requests consists of a list of all dependent data. A sensor node, upon joining a logical group, transmits dependent data to the basestation. The dependent data is automatically retransmitted whenever the dependent data is updated on the sensor node.

The collective function, upon receiving dependent data from the logical group members, operates over the received data. Since each group member continually transmits updated values, the collective function is automatically re-evaluated with new data. This automatic transmission and evaluation makes interacting with collective functions similar to interacting with local functions.

Collective functions allow users to construct sophisticated applications that require many-to-one communication. Unlike existing programming languages, all communication in Tables is implicit and specified by the interaction of pivot tables and functions.

### Chapter 3. Tables

#### Local Functions:

```
(1)
=DETECTION := 0
=if( Magnetometer > 0 )
    WY := Magnetometer * YLoc &
    WX := Magnetometer * XLoc &
    DETECTION := 1

    else DETECTION := 0
```

#### Collective Functions:

```
(3)
CentroidX := sum(ALL, WX) / sum(ALL, SENSOR)
CentroidY := sum(ALL, WY) / sum(ALL, SENSOR)
```

#### Pivot Tables:

```
(2) Data = Node ID : Metadata = DETECTION
(4) Data = CentroidX, CentroidY : Metadata = Node ID, Time, Sensor Type
```

Figure 3.8: A Tables mobile object tracking application. This application uses every element of the hierarchical group model. It first uses local functions to detect nearby objects, pivot tables to form logical groups, and collective functions to calculate the centroid of the object.

Users do not need to explicitly define transmission commands. This simplifies network programming since users do not need to be aware of explicit message handling.

Although Tables relies on spreadsheet inspired tools and lacks explicit communication, sophisticated applications, such as mobile object tracking, can be constructed with relative ease. Mobile object tracking, unlike earlier monitoring examples, requires multiple pivot tables and collective functions (Figure 3.8). The user begins by first specifying a set of local functions that determine whether a sensor node is within range of the object. Upon detecting the vehicle, a DETECTION variable and weighted locations are assigned.

After applying these functions to the sensor network, the user constructs a pivot table specifying that the DETECTION values be placed along the sheet axis. This allows all sensor nodes that have detected an object within range to share the same sheet (Figure 3.9). The user then specifies the centroid functions in the “DETECTION = 1” sheet This

### Chapter 3. Tables

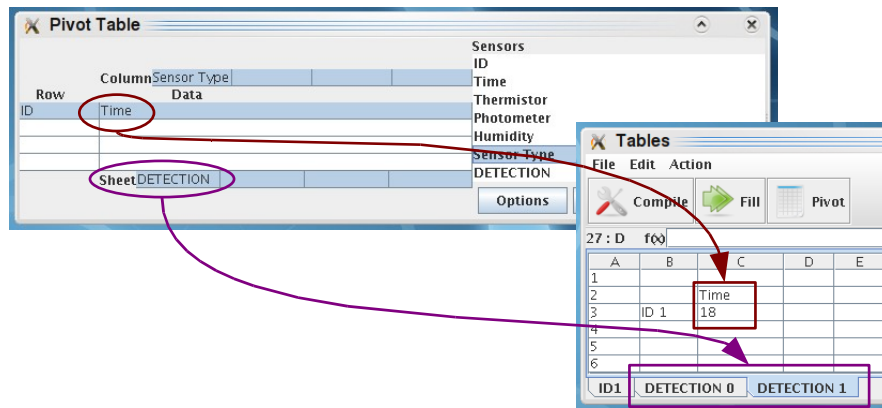


Figure 3.9: A screenshot of using the sheet pane to organize the sensor network into logical groups. By clicking-and-dragging the DETECTION value onto the sheet pane, sensor nodes organize themselves into a group that detected an object and a group that has not detected an object. The user can also view additional data such as the time that sensor nodes detected the object.

initiates the creation of a logical group (the sensor nodes within detection range of the vehicle). The centroid function, in turn, requests the weighted location values from the group members and calculates the centroids.

After allowing the application to run over a sufficient time period, the basestation will

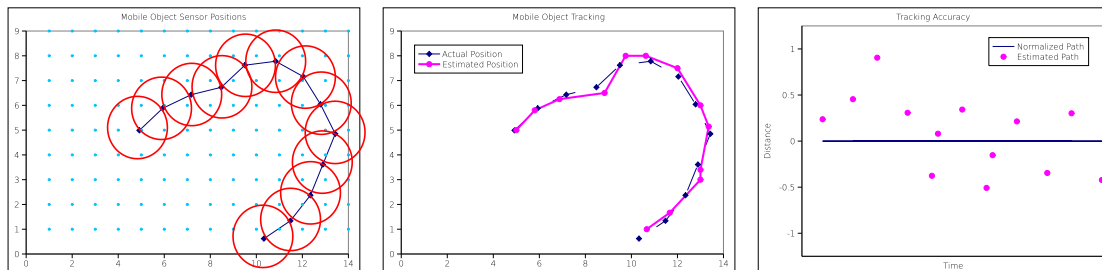


Figure 3.10: Results of the mobile object tracking application. The circles indicate which sensors are in the logical group as the object moves in a clockwise direction. The horizontal line in the right figure represents the normalized path of the object, while the points indicate how far from the actual object the centroid algorithm predicted the object to be.

accumulate several centroid locations. These locations can be retrieved by creating a final pivot table. Results from a simulated version of this application are illustrated in Figure 3.10. In the simulation, a vehicle starts at some random position and moves around in a random manner. 25 sensor nodes are placed in 1 unit increments in a grid layout. When the vehicle is within a predefined radius of a proximity sensor, the sensor registers a positive value. By using just a few functions and pivot tables, Tables allows users to create an application that normally requires explicit communication and coordination.

## **3.2 Implementation**

Tables consists of a graphical interface residing on the user's desktop and a runtime environment residing on the sensor nodes. The Tables interface is implemented as a cross-platform Java application and has been tested on various Linux desktop environments. Tables assumes that all the sensor nodes have been equipped with the Tables runtime. The Tables runtime is responsible for communication with the basestation, maintaining logical groups, and interpreting local functions. In addition, Tables assumes that a "bridge" node is connected to the basestation via USB. This bridge node simply relays packets to and from the basestation.

Tables requires the execution of local functions on sensor nodes. This is accomplished using a function interpreter on the sensor node. Functions are compiled into a simple bytecode on the basestation, propagated to the sensor nodes, and interpreted by the sensor nodes. This approach is straightforward and can be implemented in most operating system environments. However, virtual machine environments [67] or operating systems that feature dynamic binary linking [52, 33] can also be used to optimize function interpretation.

The sensor nodes were configured with 2048 bytes of dynamically allocatable memory (heap size). Although many sensor network applications are configured to use only static

### *Chapter 3. Tables*

memory, the implementation of local functions was simplified with the use of a small amount of dynamic memory. The sensor nodes were also configured to periodically collect thermistor, photometer, and humidity data and to store these results in separate data queues. During data transmission, the packet payload was limited to 40 bytes. Although the version of Mantis OS used reserves a maximum of 64 bytes for packets, much of the space is used for routing and transport headers.

Communication between the graphical interface and sensor nodes is implemented using CTP. In order to avoid interference from multiple transmitting sensor nodes, messages are transmitted separately for each sensor node. For example, pivot tables are sent separately to each sensor node. The pivot table request is not transmitted to the next node until the entire response is collected (or a timeout expires). Additionally, each sensor node that relays a packet waits a short time (200 ms in Mantis OS) before retransmission to avoid intra-path interference. In the future, a reliable transport protocol, such as Flush [63], will be integrated to handle intra-path interference.

## **3.3 Evaluation**

Applications written in Tables have a very different workflow than typical sensor network applications. Instead of constructing a monolithic application, users construct a set of pivot tables and data driven functions that communicate implicitly. Consequently, the main overhead associated with Tables is in the number of messages transmitted for the various operations. This overhead also affects user latency. For example, complex pivot tables typically transmit more data and take longer to reconstruct. In addition, the interactive workflow makes memory consumption an important issue in Tables.

In order to evaluate the relative efficiency of using pivot tables, the number of messages transmitted during a pivot table operation was measured. One of the main things that affect



### Chapter 3. Tables

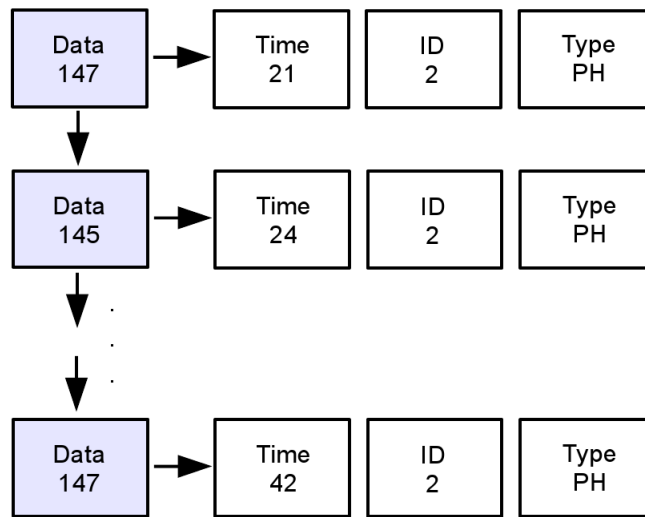


Figure 3.11: Structure of Tables pivot table replies. Each data item is described by a data identifier and includes a single data point. The data item also includes a list of metadata items. These metadata items are, in turn, described by a data identifier and data point.

the number of messages for a pivot table response is the size of the data queues stored on the sensor nodes. Each set of data items (such as thermistor data) are stored in separate data queues. Upon receiving a pivot table requesting specific data, the sensor node will fetch the entire data queue. Additionally, each data item in the queue is tagged with the relevant metadata item (Figure 3.11). Since each data item may contain a different number of metadata items, Tables must, at times, explicitly transmit the number of metadata items attached to each data item. However, in the case that all data items have the same number of metadata descriptions, Tables will omit this information.

The sensor node data queue sizes were increased from 5 to 50 in increments of 5. For each data queue size, a pivot table was constructed that requested all sensor data organized by ID, time, and sensor type. The total size of this pivot table was 11 bytes. For 50 data items, Tables transmitted between 34 and 38 messages, depending on whether the metadata description count was sent for every data item list (Figure 3.12). Currently, redundant metadata items are not compressed and hence account for the number of messages. Com-

### Chapter 3. Tables

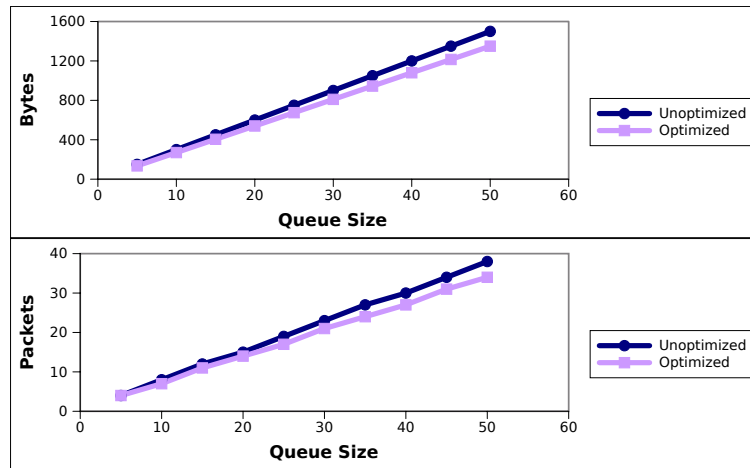


Figure 3.12: Message transmission size with respect to the data queue size. The unoptimized data assume the number of metadata associated with each data element may be a different length, and include an explicit list length for each data element. The optimized version reduces the message size when all the metadata lists are the same size.

pressing the metadata descriptors is a subject of future work.

For user latency measurements, latencies incurred using the USB and CTP communication backends were measured while varying the sensor node data queue size. For CTP, the node was placed one hop away from the bridge. Overall, the latency incurred by the CTP backend is much greater than the USB backend (Figure 3.13). This is partially be-

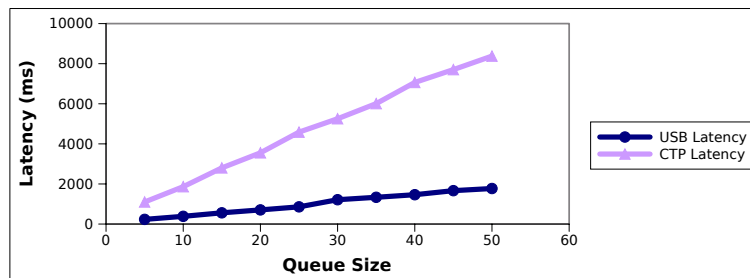


Figure 3.13: User latency with respect to the data queue size. Tables uses two communication methods: USB and a wireless routing protocol (abbreviated CTP). The data queue size determines the number of data elements transmitted back to the basestation. As such, the latency increases for both communication methods.

### Chapter 3. Tables

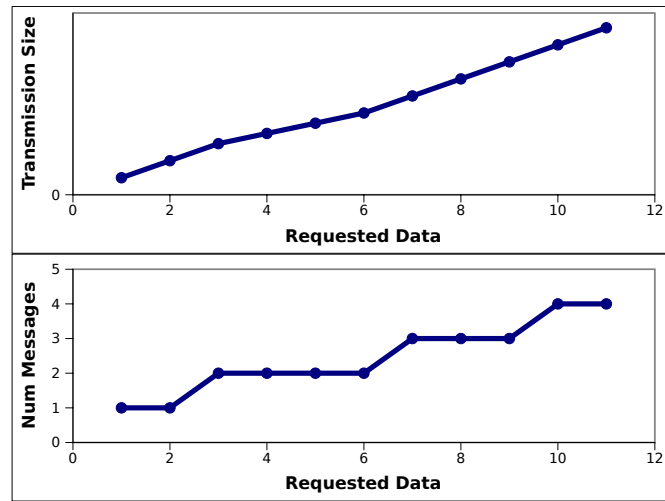


Figure 3.14: Number of messages transmitted with respect to the number of requested data. The user increases the number of requested data by filling the pivot table with the names of more and more data. Since data is tightly packed into as few messages as possible, the number of packets transmitted follows a step-wise function.

cause each CTP transmission must delay a short period of time before transmitting the message to avoid intra-path interference (200 ms in Mantis OS). In total, the user waits approximately 8 – 10 seconds to receive data when the queue is set to the maximum size of 50 using the CTP backend.

In order to measure the effect of pivot table complexity on the number of message transmissions, the number of transmitted messages was recorded while varying the number of metadata items in a pivot table. For this test, the queue size was set to 5. Pivot tables were iteratively created with each iteration increasing the number of requested metadata from 1 to 11 items. Overall, an increase in every 2 to 3 metadata items resulted in an additional packet. For the most complex pivot table (with 11 metadata items) the number of transmissions increased from a single packet to 4 packets (Figure 3.14). Although this increase is potentially large (especially for recurrent pivot tables), most pivot tables are expected to request very few metadata items. The actual pivot table always took a single packet (less than 17 bytes for a pivot table with 11 metadata items).

### Chapter 3. Tables

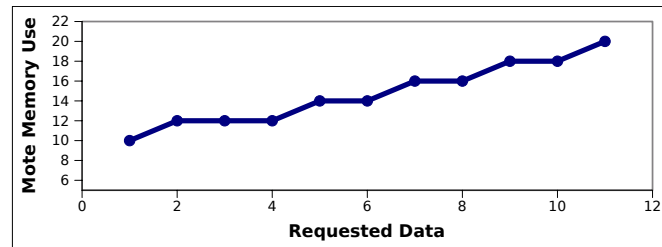


Figure 3.15: Sensor node memory consumption with respect to the amount of requested data. Requests do not consume very much memory even for large requests.

Besides network measurements, the sensor nodes are also restricted by the amount of available memory. The amount of memory the sensor node consumed for each data queue size was measured. The sensor node consumes 284 bytes for small queue sizes and consumes up to approximately a kilobyte for 50 values (half of the total heap size). Since other operations, such as responding to pivot tables, also consume memory, users should not configure much larger data queue sizes.

When a sensor node receives a pivot table request, the node must allocate memory to store the request and to construct the reply. Although the memory allocation is transient (the memory is released immediately after transmitting the reply), every data item and metadata item associated with the reply is reallocated. This is necessary since the reply structure can be complex. However, even with these additional allocations, memory increases modestly from approximately 10 to 20 bytes (Figure 3.15).

Another major source of memory consumption are local functions stored on the sensor nodes. For each local function, the sensor node determines the data dependencies and registers the function with the appropriate data queues. Upon sampling some data, all functions that depend on that data are interpreted. In order to investigate the memory consumption with respect to the number of local functions, up to 9 vector functions with a single dependency were stored. 9 independent vector functions that do not share any data dependencies were also stored. As Figure 3.16 shows each vector function con-

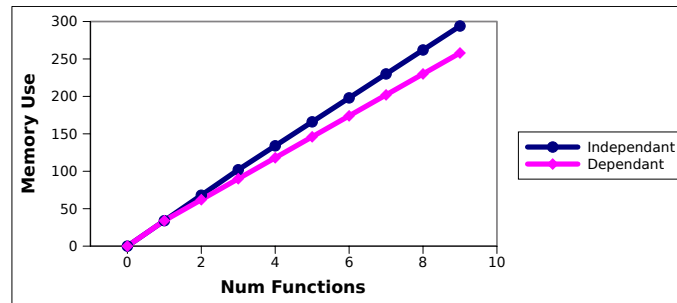


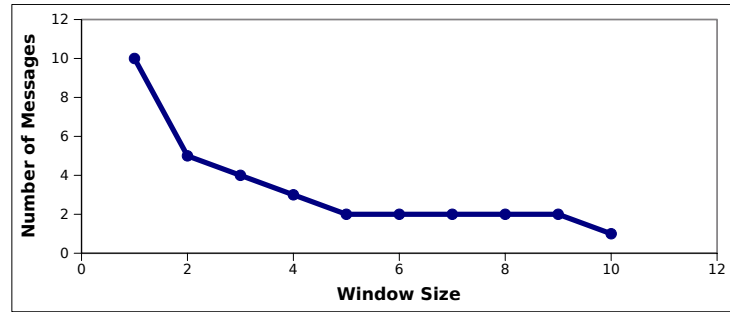
Figure 3.16: Sensor node memory consumption with respect to the number of functions stored on the node. Independent functions are functions that do not share any data dependencies. As such, these functions consume more memory than dependent functions, which all share the same data dependency.

sumes 34 bytes if it contains a new data dependency and 28 bytes if using an existing data dependency. Most real applications use a combination of both types of functions.

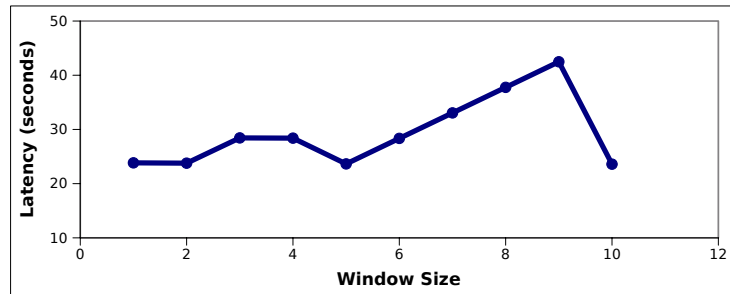
Currently for collective functions dependent data is transmitted from the sensor nodes to the basestation. Normally a sensor node would transmit each new data element. However, doing so may transmit a large number of messages since each data element monopolizes an entire packet. In order to conserve the number of messages transmitted, we investigated the effects of batching the dependent data before publication. For any given publication window size,  $n$ , the sensor node collected  $n$  data values before transmitting. Overall this significantly decreases the number of messages transmitted (Figure 3.17(a)).

However, latency experienced by the user is not uniformly decreased. Although latency decreases due to fewer messages, latency may also increase due to unnecessary data sampling. Since most collective functions operate over multiple data (say 10), the collective function must wait for the right number of data before it can be evaluated. However, transmitting more data than necessary increases the initial latency since the sensor node must spend additional time sampling and filling up the publication window. As Figure 3.17(b) illustrates, the initial latency can increase for large window sizes. However, after the initial evaluation, the additional data is counted towards the next evaluation.

### Chapter 3. Tables



(a) By modestly increasing the window size, the user can significantly reduce the number of messages transmitted, conserving the amount of energy.



(b) Latency decreases with fewer messages, but can also increase since the sensor node must wait for additional data samples to fill the publication window even if the collective function does not require it.

Figure 3.17: Number of transmitted messages and latency with respect to the publication window size.

Although applications written in lower level languages may be more efficient, Tables targets users that may have difficulty constructing such optimized applications. Instead, Tables offers an interactive programming environment that allows users to construct complex applications using simple tools at the cost of modest overhead. Additionally, future implementations of Tables may employ optimizations, such as optimizing pivot table requests, to further reduce the overhead.

## 3.4 Discussion

Although Kensho simplifies sensor network programming by directly implementing the hierarchical group model, the barriers to programming are still relatively high for many users. Tables, a graphical programming environment for sensor networks, lowers the barrier to entry using a variety of spreadsheet-inspired tools. By combining pivot tables with various functions, users can specify both local and collective computation. This is made possible by modeling the communication and computation tasking according to the hierarchical group model. Unlike Kensho, Tables employs data-driven functions as the computational unit. This simplifies programming and adheres to expected spreadsheet semantics. Communication between local and collective functions is implicit with the necessary data being published automatically. Logical groups are also defined implicitly by using pivot tables to organize data along the sheet dimension.

Currently Tables allows users to specify node-specific local functions. Although the hierarchical group model does not explicitly define such behavior, this was necessary in order to preserve simple spreadsheet semantics. This behavior can be emulated within the hierarchical group model by treating node ID as a type of sensor data. Logical groups consisting of a single node can then be created using this unique sensor data. Although not ideal, this demonstrates that the hierarchical group model can be used to replicate many node-centric models.

Although Tables implements its own version of the hierarchical group model, it is possible to also use Kensho as the underlying communications and tasking library. Much of the protocol remains the same due to their common computational model. However, Kensho explicitly assumes that the underlying computational unit is a Mantis OS thread. Although threads can be adapted for interpreting Tables functions, this may introduce additional memory and computational overhead.

## **Chapter 4**

# **Applications of the Hierarchical Group Model**

Kensho, and the underlying hierarchical group model, not only characterizes standard sensor network applications, but also aids in the construction of more complex sensor network software. Two examples include sensor network management software and privacy-preserving algorithms that can be integrated into existing applications. Both sets of software, programming interfaces and privacy algorithms, are important emerging aspects in sensor networks. As sensor networks become more prevalent in both academic and everyday use, users will expect intuitive tools to interface with sensor networks. Users will also expect sensor network applications to conform to strict privacy requirements. Unlike standard applications, these examples may exhibit more complex communication patterns, and by conforming to the hierarchical group model, they can be made to integrate into other sensor network scenarios.



## 4.1 FUSN

As sensor networks become more widely used by the public, new tools will need to be developed that allow these users to easily manage sensor networks. While tools like Tables aid in the development of new applications, casual use and inspection may require still simpler tools. FUSN (pronounced “fusion”) is a framework to construct filesystem interfaces for sensor networks. FUSN is not a single filesystem interface that users interact with. Instead, FUSN consists of an API and associated set of mechanisms that allow system developers to implement virtual filesystems similar to the Unix */proc* and */sys* filesystems. FUSN employs FUSE [3] to create a POSIX-style filesystem and includes mechanisms to automatically handle all communication between the sensor network and filesystem host. By exposing the sensor network as a virtual filesystem, system developers retain the freedom to construct complex applications, while ensuring that end-users are able to manage and interact with the sensor network in a familiar manner.

End users interact with FUSN-based filesystems by mounting the filesystem, and using common tools such as *ls*, *cat*, and *echo* to view and update data, organize groups of sensors, and control access to data. Similarly, other software can use existing file I/O libraries to interact with the sensor network. This allows users to construct prototype sensor network applications in more familiar programming environments such as *Matlab*. Using FUSN debugging, application prototyping, and file viewing become commonplace. Unlike previous tools that were designed specifically for sensor networks [27], FUSN allow users to leverage a large body of existing software originally designed for filesystems. Also, FUSN does not limit interaction to existing tools. As new tools for filesystems develop, they can be applied to sensor networks as well.

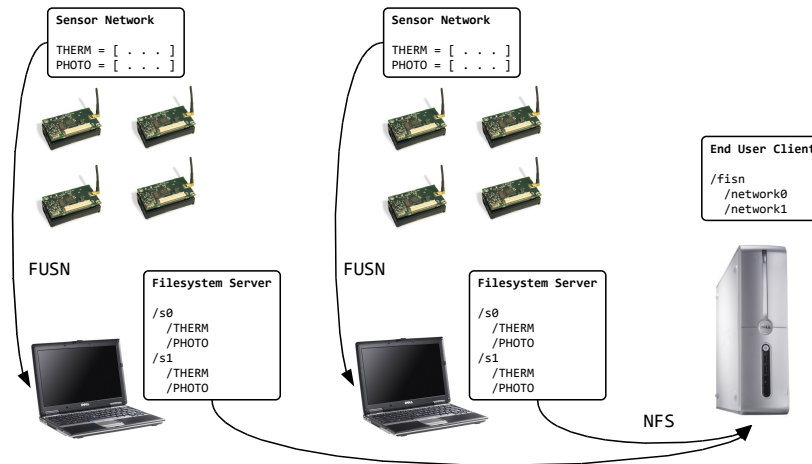


Figure 4.1: The FUSN architecture. FUSN consists of a filesystem server that translates filesystem commands into sensor network operations. The filesystem server, in turn, may export additional networking services. Sensor nodes collect data and communicate with the filesystem server.

### 4.1.1 Architecture and Application Programming Interface

The FUSN architecture consists of two main components: the filesystem server and the FUSN runtime executing on the sensor nodes. These components are organized in a tiered fashion to facilitate implementation using Kensho (Figure 4.1). Currently, the sensor network must be physically contiguous and it communicates with one filesystem server. The filesystem server is assumed to be more powerful (typically a PC-class device) than the sensor nodes and handles most of the complicated filesystem processing. Sensor nodes, on the other hand, are assumed to be mote-class devices that can only handle minimal processing. This tiered organization is used in other sensor network architectures [49] and provides several benefits, including reduced code complexity and potential memory savings on the sensor nodes. This tiered organization also conveniently matches the hierarchical group computation organization, simplifying implementation.

The filesystem server is primarily responsible for presenting a filesystem interface to

Chapter 4. Applications of the Hierarchical Group Model

Programming Interface	Operation Description
linked_list list_data()	List names of data on sensor node
linked_list read_data(name)	Reads content of specified data item
uint8_t write_data(name, buffer)	Writes new content to the specified data item
uint8_t delete_data(name)	Deletes the specified data item

Table 4.1: Summary of the FUSN API. The user can construct new filesystem interfaces by implementing these functions. Communication between the sensor nodes and filesystem server is handled automatically.

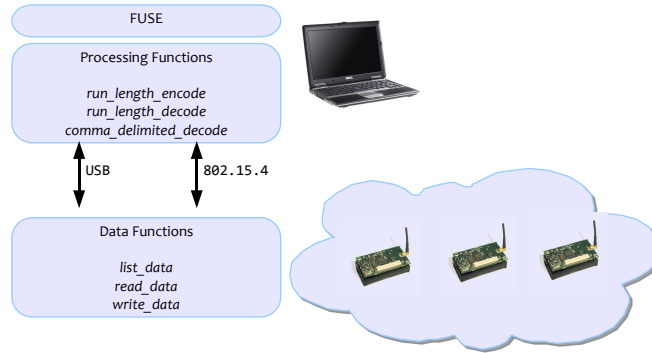


Figure 4.2: FUSN compilation architecture. Processing functions execute on the filesystem server while data functions execute on the sensor nodes. This tiered organization matches most sensor network application characteristics and simplifies implementation.

the user. After the user initiates a specific filesystem command, the supplied path is resolved to a sensor node and a file name. This file name is then sent to the sensor node with the relevant command. In turn, the FUSN runtime on the sensor node responds to the command by supplying the requested data. Otherwise, applications executing on the sensor nodes do not directly interact with the FUSN runtime. Ultimately, the filesystem server retrieves the data and converts it to the necessary filesystem data structures. Besides presenting the filesystem interface, the filesystem server may also export services such as *NFS*. This enables users to manage a confederation of sensor networks by adding a internet-level tier to the architecture.

## Chapter 4. Applications of the Hierarchical Group Model

```
struct fusrn_proc_info pre_processor(struct linked_list* input,
                                   struct fuse_file_info* fi,
                                   struct linked_list* data)
{
    input_data = (struct fusrn_output_buffer*)input->head->data;
    buf = (char*)input_data->data;

    for(cmd_size = 0, i = 0; i < input_data->size; ++i) {
        if(buf[i] != ' ' && buf[i] != '\n' && buf[i] != '\t')
            command[cmd_size++] = buf[i];
    }
    command[cmd_size++] = '\0';

    input_data->size = cmd_size;
    memcpy(input_data->data, command, cmd_size);

    info.data = input;
    info.file_info = fi;

    return info;
}
```

Figure 4.3: FUSN pre-processing function. Pre-processing functions execute on the filesystem server before the data is sent to the sensor network. This can be used to perform data conversion tasks, such as sanitizing data.

```
struct linked_list* post_processor(char* file,
                                  struct linked_list* data)
{
    for(temp = data->head; temp != NULL; temp = temp->next) {
        user_data = (struct fisn_data_vector*)temp->data;
        sprintf(itoa_string, "%d", *(uint16_t*)user_data->data);

        user_data->size = strlen(itoa_string);
        memcpy(user_data->data, itoa_string, user_data->size);
    }

    return data_items;
}
```

Figure 4.4: FUSN post-processing function. Post-processing functions also execute on the filesystem server over data recently received from the sensor network. This can be used to convert raw sensor data into a more appropriate format for the user.

## Chapter 4. Applications of the Hierarchical Group Model

In keeping with the tiered organization, FUSN provides a split API that allows developers to specify which functionality executes on the sensor network and which functionality executes on the filesystem server. As illustrated in Figure 4.2, “data-oriented” functions execute on the sensor nodes, while “processing” functions execute on the filesystem server. This enables the filesystem developer to optimize the interface for reduced memory consumption and message transmission.

The data-oriented functions specify where and how the data presented to the user is generated (Table 4.1). Currently, the user is expected to implement the *list\_data* and *read\_data* functions (for a read-enabled filesystem) and *write\_data* and *delete\_data* (for a write-enabled filesystem). The *read\_data* function accepts the name of the file the user has requested and returns a list of data. Similarly, the *list\_data* function simply returns a list of file names to expose to the user. It should be noted that the data constructed in these functions are not restricted to raw sensor data and can include application specific data. For example, the filesystem developer can choose to export memory consumption data using these functions. Also, communication between the filesystem server and the sensor network is automatically handled; the developer simply constructs the list of data to be transmitted.

The *write\_data* function accepts the name of the file and the data to be written. It also includes parameters to accommodate truncations and appends. This data can be stored on the sensor node or be used by the filesystem developer to control devices. Currently FUSN does not provide interfaces to explicitly control file attributes or permissions. File attributes, such as file sizes and modification dates, are calculated automatically by the filesystem server. Similarly, file modes and permissions are stored and handled automatically by the filesystem server.

In order to provide a flexible way to implement functionality on the filesystem server, FUSN provides a pre-processing and post-processing API. Pre-processing functions are used to transform input provided by the end user before sending to the sensor network.

## Chapter 4. Applications of the Hierarchical Group Model

This can be used for example to sanitize or compress input (Figure 4.3). Similarly, post-processing functions are used to transform data provided by the sensor network before presenting it to the user. This can be used for example to calibrate raw sensor values or transform a list of integers into a comma-delimited text file (Figure 4.4).

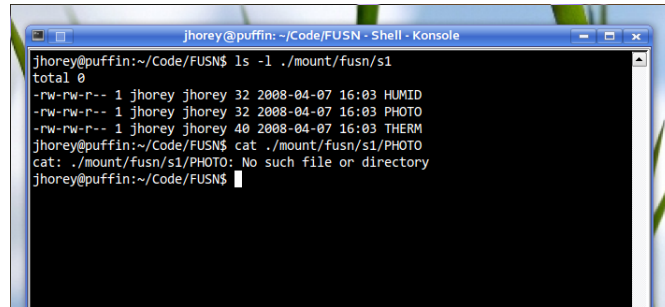
Processing functions can also be used in combination. For example, one of the filesystem interfaces developed employs a parsing pre-processor and an equivalent unparsing post-processor. This allows the filesystem interface to export the illusion that the sensor nodes stores the unparsed data while reducing the memory consumed by the sensor nodes. Finally, FUSN provides support to stack pre- and post-processing functions, enabling multiple data transformations.

Using both the data-oriented functions and processing functions, the user is able to create a wide array of different filesystem interfaces. However, FUSN currently has two major limitations. First, FUSN does not support the creation of links (usually created using the *ln* command). Besides complicating the possible semantics of the filesystem, links are often not used in file I/O operations. Second, FUSN assumes that directories represent either sensor nodes or groups of sensor nodes. Although it is possible to use pre-processing functions to emulate a data directory, such as “directory containing sensor nodes equipped with thermistors”, this is currently cumbersome.

### 4.1.2 Implementation

The FUSN runtime is implemented on the TelosB motes using the Kensho library. The Kensho runtime is used for communication between the sensor node and the FUSN base-station and is also used for local tasking. Currently, advanced features such as logical groups are not used. Each sensor node is loaded with an application statically linked with the FUSN runtime. This runtime is executed as a local task, allowing the main application logic to remain simple. The FUSN runtime responds to messages from the filesystem

## Chapter 4. Applications of the Hierarchical Group Model

A terminal window titled "jhorey@puffin: ~/Code/FUSN - Shell - Konsole" showing the output of a file system listing command. The output shows three files: HUMID, PHOTO, and THERM. The user then attempts to cat the PHOTO file, which results in an error message: "cat: ./mount/fusn/s1/PHOTO: No such file or directory".

```
jhorey@puffin:~/Code/FUSN$ ls -l ./mount/fusn/s1
total 0
-rw-rw-r-- 1 jhorey jhorey 32 2008-04-07 16:03 HUMID
-rw-rw-r-- 1 jhorey jhorey 32 2008-04-07 16:03 PHOTO
-rw-rw-r-- 1 jhorey jhorey 40 2008-04-07 16:03 THERM
jhorey@puffin:~/Code/FUSN$ cat ./mount/fusn/s1/PHOTO
cat: ./mount/fusn/s1/PHOTO: No such file or directory
jhorey@puffin:~/Code/FUSN$
```

Figure 4.5: FUSN failure model. FUSN will report a non-existent file error if the sensor node is disconnected during a read or write operation.

server, executes the user-specified data functions, and transmits the data back to the filesystem server.

FUSE, a user-space filesystem module that simplifies the construction of Unix filesystems, is used to construct the filesystem presentation. FUSE provides a user-level API similar to the Posix file I/O API. By constructing functions that conform to this API, users are able to construct full-fledged filesystems. Like native filesystems, FUSE filesystems can be mounted anywhere in the filesystem tree; programs designed to interact with the filesystem do not distinguish between native and FUSE filesystems. FUSE is currently available for Linux, FreeBSD, and Mac OS X.

The FUSE filesystem server communicates and tasks the sensor network using the Kensho communication methods. When a particular filesystem request is made, the relevant FUSN function is invoked and a message is constructed that is “pushed” to the sensor network. The sensor nodes are locally tasked with the relevant FUSN local runtime threads and responds to the messages. These responses are, in turn, “published” back to the filesystem server. However, unlike other Kensho applications, FUSN does not employ collective tasks. This is primarily because FUSE itself is an event-driven system where user actions (such as typing in *ls*) invokes certain event handlers (the FUSE functions). However, the Kensho functions can be used even in non-threaded environments. After the

## Chapter 4. Applications of the Hierarchical Group Model

```
function readsensors(path)
    while(true)
        sensor_light = dlmread(strcat(path, '/s0/PHOTO') )

        plot(sensor_light, 'mo')

        pause(2)
    end
end
```

Figure 4.6: Matlab script using FUSN to interact with a sensor node. The script is able to read the photometer value from the sensor node by issuing a standard *dlmread* call. FUSN automatically transmits the relevant data and formats it in a comma-delimited form.

FUSE functions perform a “collect” to receive the data, the appropriate FUSN processing functions are applied.

In the case that packets are not received by the filesystem server, FUSN translates networking errors into appropriate file I/O errors. Filesystem failure and recovery has been extensively studied [88, 65, 58], and many of these ideas may be applicable for sensor networks. However, currently FUSN employs a simple failure and availability model. For a directory listing command, FUSN will simply not list the file. For data retrieval commands, FUSN will report a file with zero size if the file has already been opened, otherwise it will report a non-existent file error (Figure 4.5).

### 4.1.3 Filesystems

In order to evaluate the efficacy of the FUSN API, two different filesystems were implemented. Both filesystems specify pre and post-processing steps to reduce data communication and allow users to control the behavior of the application using *write* commands. The first filesystem exposes each sensing device (photometer, thermistor, and humidity) and LED device as a file. Each sensor node continually collects sensor data and stores the data in a limited size buffer. The basic filesystem employs a post-processing function that



<b>Task Command</b>	<b>Task Operation</b>
sample(IN, OUT, SAMPLES, RATE)	Collects input and stores in output
classify(IN, OUT, OP, THRESH)	Classifies input and store results in output
stat(IN, OUT, OP, SAMPLES)	Performs statistics and store results in output
stamp(IN1, IN2, OUT)	Appends values and stores the result in output

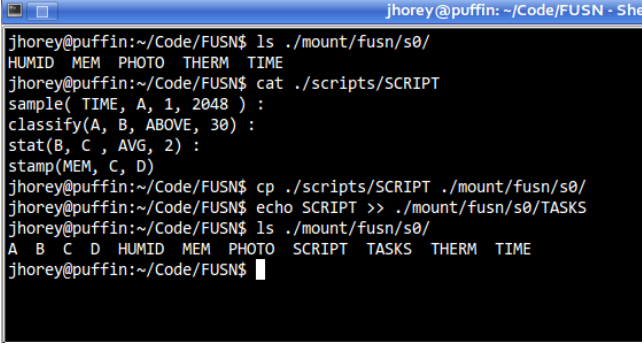
Table 4.2: Functions supported by the FUSN task interpreter. Functions include methods to sample sensor data, perform statistics on the data, and classify the data. Users are able to task the sensor node by writing these functions into a standard text file.

transform the list of integer values into a comma-delimited text file. This conversion helps users to view data in a form that most data analysis programs understand. For example, the user is able to use the *Matlab* `dlmread` function without modification to read in the sensor data (Figure 4.6).

The basic filesystem also allows users to interact with the LED devices by reading and issuing appropriate commands. Upon starting the basic filesystem, all the LEDs are initialized to the *off* state. In each LED file, the user will be presented with a text file containing the string *OFF*. By writing a command, such as *ON* or *TOGGLE* to the appropriate LED file, the user is able to control the state of the LEDs. The basic filesystem employs a pre-processing function that automatically removes whitespaces, and adds null-terminators when necessary. Although simple, this basic filesystem is functional as a data logging mechanism and serves to help filesystem developers construct more complex filesystems.

Besides the basic filesystem, a more sophisticated command-and-control filesystem was constructed that includes a simple task interpreter. These tasks are inspired by the Tenet tasking library [49]. Like the original Tenet tasking library, the command-and-control filesystem provides a set of simple functions that can be chained together to perform a particular task. Currently four functions (Table 4.2) are provided, a subset of the original Tenet tasks. Each function accepts at least one input file and an output file. It performs a specific operation over the input and places the result in the output. For example,

## Chapter 4. Applications of the Hierarchical Group Model



```
jhorey@puffin: ~/Code/FUSN - She
jhorey@puffin:~/Code/FUSN$ ls ./mount/fusn/s0/
HUMID MEM PHOTO THERM TIME
jhorey@puffin:~/Code/FUSN$ cat ./scripts/SCRIPT
sample( TIME, A, 1, 2048 ) :
classify(A, B, ABOVE, 30) :
stat(B, C, AVG, 2) :
stamp(MEM, C, D)
jhorey@puffin:~/Code/FUSN$ cp ./scripts/SCRIPT ./mount/fusn/s0/
jhorey@puffin:~/Code/FUSN$ echo SCRIPT >> ./mount/fusn/s0/TASKS
jhorey@puffin:~/Code/FUSN$ ls ./mount/fusn/s0/
A B C D HUMID MEM PHOTO SCRIPT TASKS THERM TIME
jhorey@puffin:~/Code/FUSN$
```

Figure 4.7: Screenshot of a user tasking and debugging a script using FUSN. The user is able to copy a text file containing the script to the relevant sensor node directory. After entering the name of the script to the TASKS file, the sensor node begins to interpret the script. The task then generates intermediate files (A, B, C, and D) that contain debugging information.

the *sample* task continuously reads the input file at a specified rate and collects a specified number of samples. It then takes these samples and places them in the output file. This can be used to sample various sensor devices. Besides *sample*, none of the other functions execute continuously.

Users invoke the task interpreter by creating a file listing the functions that constitute the task. The user must also append the name of that file to a file called “TASKS”. The TASKS file simply lists all the tasks the sensor node must execute. These files must reside in the sensor node directory that is supposed to execute those tasks. All files can be created using a normal text editor or using *echo* and *cp* commands (Figure 4.7). The task interpreter monitors the task control file for new tasks and automatically begins interpreting the task functions. Currently, each task is interpreted as a separate Mantis thread. As tasks are interpreted, users can view the intermediate input and output files used by the tasks.

Figure 4.8 illustrates the dynamic memory consumption of a complex task containing all four functions. Each intermediate file contains the four most recent values. This aids in debugging since the user has access to a history of values. The memory consumption increases after the task file is loaded and begins execution. The task only begins collecting

## Chapter 4. Applications of the Hierarchical Group Model

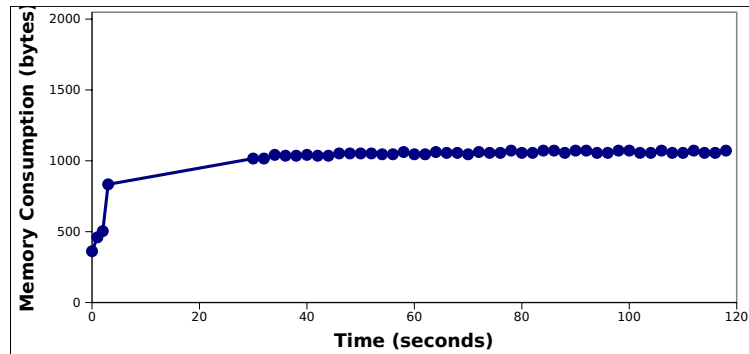


Figure 4.8: Memory consumption of a single task file over time. The sensor node initially allocates memory for storing the task and generating intermediate data structures. Afterwards, the task interpreter maintains a steady state.

Task Command	Text Size	Binary Size
sample(TIME, A, 1, 2048)	22 bytes	14 bytes
classify(A, B, ABOVE, 32)	25 bytes	15 bytes
stat(B, C, AVG, 3)	19 bytes	13 bytes
stamp(MEM, C, D)	16 bytes	11 bytes

Table 4.3: Static memory consumption of the tasking functions before and after compression. The uncompressed function stores the entire function verbatim, while the compressed function stores an intermediate representation that remove superfluous tokens. This results in a decrease in memory use of over 20%.

data after 30 seconds, after which the memory consumption remains relatively constant. Since the goal of the tasking filesystem is to demonstrate how to integrate FUSN with a programming environment, the system was not optimized for minimal memory consumption or runtime overhead.

In order to lower static memory usage and minimize the computational overhead, the command-and-control filesystem employs both a pre-processing parsing step, and a post-processing unparsing step. In the parsing step, the filesystem server tokenizes and parses the task text file. The result is then sent to the sensor node for evaluation, saving the sensor node from parsing the text itself. On average, this reduces static memory consumption by

<b>Program</b>	<b>Text Size</b>	<b>Data + BSS Size</b>
blink_led	13372 bytes	1592 bytes
sense_and_forward	15030 bytes	1702 bytes
basic filesystem	33804 bytes	6298 bytes
tasking filesystem	36276 bytes	6338 bytes

Table 4.4: Binary size of Mantis OS applications and different FUSN filesystems. Although FUSN filesystems consume more static text and data than typical Mantis OS applications, complex filesystems only modestly increase the text and data size.

approximately 35% (Table 4.3). Similarly, when the user requests to view the task file, the sensor node transmits the compressed parsed form. The host automatically re-creates the textual form of the file and displays it to the user. This is all done automatically; from the user’s perspective the text files appear to reside on the sensor nodes.

#### 4.1.4 Evaluation

FUSN filesystems, unlike typical applications, can serve multiple purposes (such as data sampling or tasking). As such, these filesystems tend to have larger binaries, which may affect future work with respect to dynamic binary linking. However, as Table 4.4 illustrates, the text size of these filesystems is only modestly more than that of a simple “sense and forward” program packaged with Mantis OS. However, FUSN filesystems do consume more global data than typical Mantis OS applications. This is something that ultimately affects the total amount of memory available to applications (examined later in this section). However, constructing more advanced filesystems, such as the tasking filesystem, does not substantially increase the text or data size over simple FUSN filesystems.

Another key overhead associated with FUSN is sensor node memory consumption and message cost. In order to evaluate these costs, latency and memory measurements were gathered using the USB communications backend. This was done to disentangle

## Chapter 4. Applications of the Hierarchical Group Model

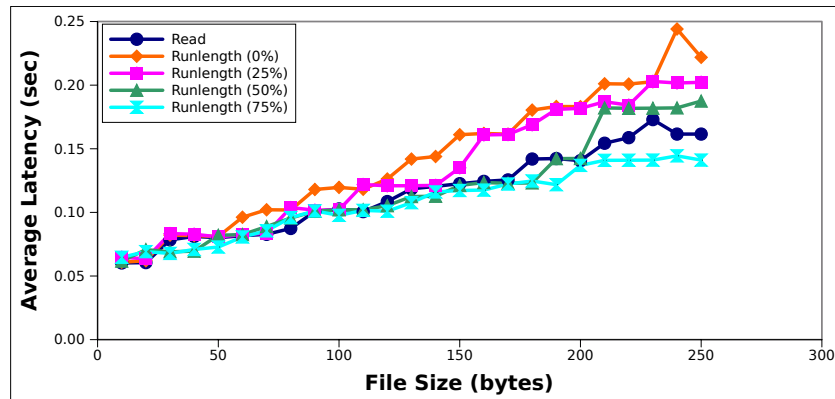


Figure 4.9: The latency of opening and reading a file with respect to file size. The file was compressed in various proportions using runlength encoding. Compressing the file less than 50% resulted in higher latency due to the time to compress the data.

the overhead associated with the wireless protocol from that of FUSN. From the user's perspective, the CTP and USB backends only differ in the perceived latency of certain I/O operations.

Besides the basic and tasking filesystems, a *runlength* filesystem was implemented to study the effects of computation and communication delay on end-user latency. The runlength filesystem generates and encodes a list of values at specified compression rates using runlength encoding. The runlength encoding algorithm replaces a set of contiguously identical values with a preamble specifying the number of times the value appears along with the value itself. This encoding scheme works best in scenarios where sensor values may not change for several collection periods. The encoding takes place on the sensor node while decoding takes place on the filesystem server as a post-processing function. A simple *read* filesystem that simply generates data but does not employ runlength encoding was also developed for comparison.

## File Latency

Using the runlength and read filesystems, the latency of opening, reading, and closing a file was measured. The data sent by the sensor nodes consisted of a set of integer values compressed at various levels. The host then decompressed and transformed these values into a set of a comma-delimited numbers using a post-processing function. All tests were performed on the host executing a simple Ruby script. This script consists of an *open* command followed by a *readline* command to read the data.

As Figure 4.9 illustrates, latency for the smallest files (containing 10 values) for all filesystems was approximately 6 ms. The latency as the number of integer values in the file is increased (increments of 10 up to 250 values) was also measured. Each data point is an average over 10 independent trials. Error bars showing standard deviation at each data point were very small (tens of microseconds) and are not visible.

Surprisingly, the latency of the read filesystem, without runlength encoding, was consistently lower than the runlength filesystems that encoded with low compression rates. Runlength encoding confers a latency advantage with compression rates of approximately 75%. This is because there are two primary sources of latency: the latency involved in the transmission of a packet over the medium (USB in this particular case) and the latency associated with the runlength encoding computation. For USB, the latency involved in the runlength encoding is greater than the latency in the actual message transmission. As such, the end-user does not perceive a latency reduction until the number of messages is significantly reduced.

Although the USB communications latency is currently small, I performed tests that investigated the perceived end-user latency when the communication latency is increased. This can happen, for example, when the wireless link between two sensor nodes is very poor and CTP sends multiple retransmissions. In order to measure these effects, an artificial delay in the transmission function was introduced. The sensor node was programmed

## Chapter 4. Applications of the Hierarchical Group Model

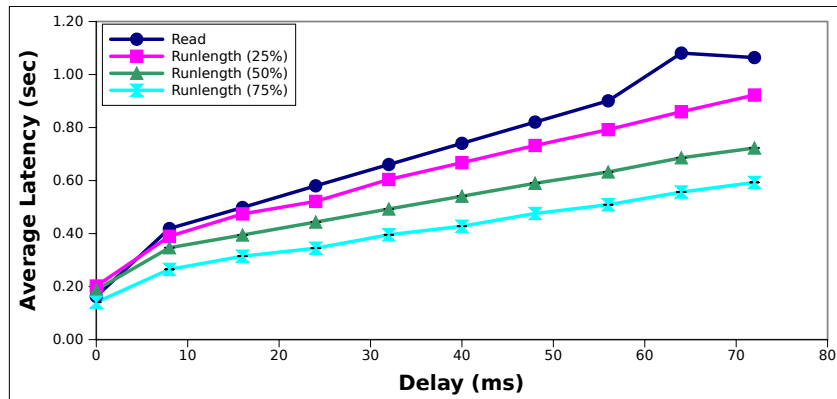


Figure 4.10: The latency of opening and reading a file as the transmission delay is increased. Transmission delay is used to emulate the effects of different communication methods. The runlength compressed files have lower latency over modest increases in transmission delay.

to transmit 250 values at specified compression rates. The artificial delay was increased from 0 to 72 ms in increments of 8 ms. The test was performed ten times and all values were averaged.

As Figure 4.10 illustrates, even a small amount of delay lowers the overall end-user latency when using runlength encoding. As expected, as the artificial delay increases, the latency difference in compression rates becomes more substantial. Using this information, filesystem developers can choose the appropriate amount of compression to use in any particular application scenario for minimal end-user latency.

Besides measuring the latency to read large amounts of data from a file, the latency to open and read a small file multiple times was measured. The read and runlength filesystems exposed a single file of 10 values that was read by the host sequentially multiple times. As Figure 4.11 illustrates, latency increases linearly with the number of times the file is read. Overall, latency increases much faster than reading a single large file with an equivalent amount of data. This is due primarily to the overhead of sending and receiving FUSN file request packets and the inability to compress a large number of values on the

Chapter 4. Applications of the Hierarchical Group Model

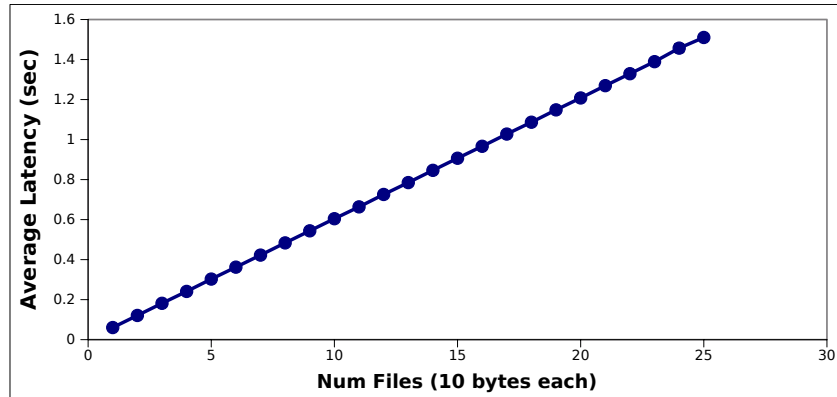


Figure 4.11: The latency of opening and reading a file with respect to number of files. The latency increases linearly since data across files are not compressed.

sensor node. From this data, it is evident that filesystem developers should preferentially use few files of larger size, rather than many files of smaller size.

Finally, the total latency overhead of common command-line programs, such as *ls* and *cat*, was measured using the *basic* filesystem. Even though the semantics of each program are simple, each program may make multiple filesystem calls, increasing their total

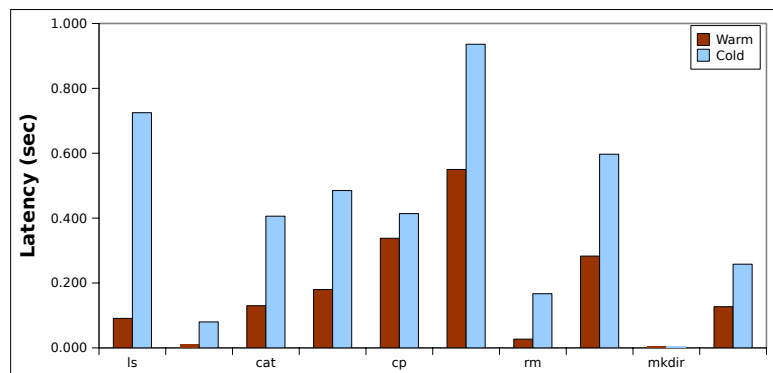


Figure 4.12: End-user latency experienced by the user for several command-line tools. The *cold* bar times are recorded after the first invocation of the command-line program. Afterwards, the file metadata is cached. Subsequent invocations of the command-line program produces the *warm* bars. Most command-line programs take less than a second to complete.



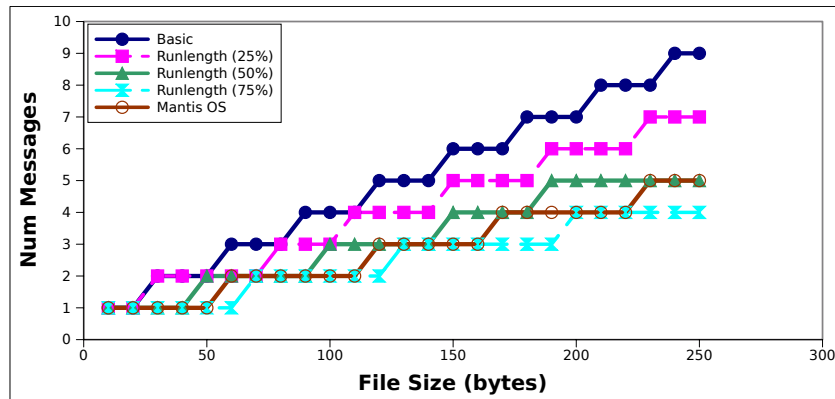


Figure 4.13: Number of messages transmitted by a sensor node with respect to file size and different compression levels. Compressing the data by 25% reduces the number of messages transmitted by 1. The number of messages transmitted is a step-function since the data packet may not contain the maximum amount of data.

latency. These filesystem calls are often redundant (multiple calls to get the file attributes). Although the filesystem server caches much of the metadata information, FUSN can be further optimized in this regard.

For these tests, both the latencies before the metadata was cached (*cold* latency) and after the metadata is cached (*warm* latency) were measured. The cold latencies were measured by mounting the filesystem and immediately executing the command-line program. Similarly, the warm latencies were measured by first performing an *ls* in the sensor node directory before executing the command-line program. Performing an *ls* in a directory requests the necessary metadata for each file, allowing the filesystem server to cache the relevant information. Overall, command-line programs that retrieve file attributes often (such as *ls*) have the highest difference between cold and warm latencies (Figure 4.12). For most programs the cold latency is about twice the warm latency, although most programs complete in less than a second even with cold caches.

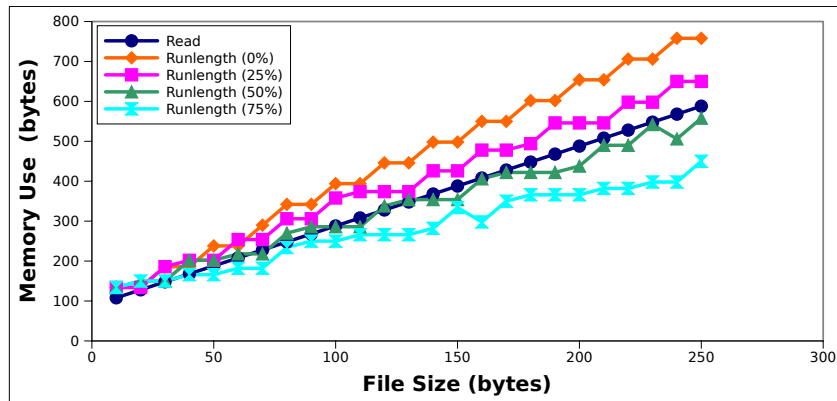


Figure 4.14: Dynamic memory consumption with respect to file size for multiple filesystems. For small file sizes and low compression rates, the compressed filesystems may consume more memory compared to a non-compressed filesystem due to the structure of runlength encoding.

### Sensor Node Overhead

In order to quantify the overhead associated with the sensor node, the total number of messages the sensor nodes transmit was measured during read operations along with the dynamic memory consumption of the filesystem. Besides affecting file latency, the number of messages also affects the energy consumption of the sensor node. The sensor node was equipped with both the basic and runlength filesystems. The number of integer values in the file being read was varied from 10 to 250 values in increments of 10. After a read request, the sensor node transmits all the requested values, and finally sends a completion packet indicating the number of packets the host should have received. As Figure 4.13 illustrates, the number of messages transmitted is a step-wise function of the compression rate employed by the filesystem. These steps reflect the number of messages that can fit into a single packet (a maximum of 64 bytes).

To measure dynamic memory consumption, the dynamic memory allocator was supplemented with FUSN to display the amount of allocated heap space as a file. The sensor nodes were loaded with the runlength and read filesystems, and file sizes were varied from

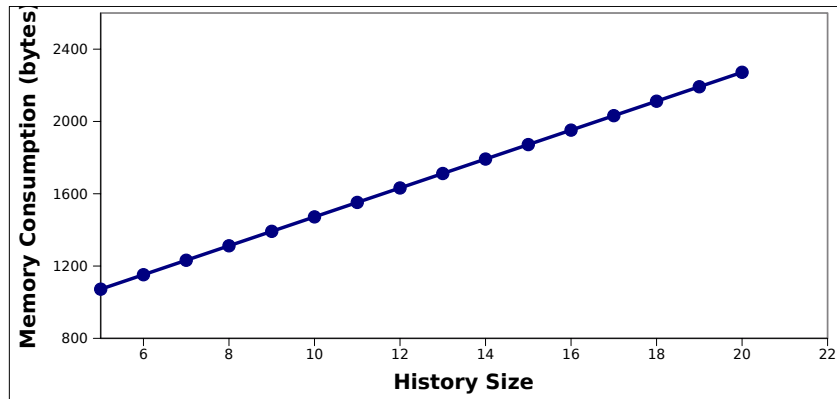


Figure 4.15: Dynamic memory consumption of an executing task with respect to the size of the intermediate files. A larger history size aids in debugging, but also increases memory usage linearly.

10 to 250 bytes. As Figure 4.14 illustrates, the runlength filesystem consumes more memory than the read filesystem at low compression rates. This is because the read filesystem allocates a single large array of the appropriate size and stores all values in that array. The runlength filesystem, however, must allocate a list of runlength preambles. Each preamble either consists of a compressed value, or an array of discontinuous values. As the compression rate is increased (to over 50%), the runlength filesystem must allocate fewer and fewer preambles and ultimately consumes less memory than the read filesystem.

Unlike the other filesystems, the task filesystem is designed to allow users to debug running programs by viewing intermediate files. These intermediate files can be configured to contain multiple historical values. For example, by issuing a read command, the user can view the  $n$  most previous values, where  $n$  is the history size. However, as Figure 4.15 shows, increasing the size of the intermediate file in the task filesystem decreases the amount of available memory for the interpreter.

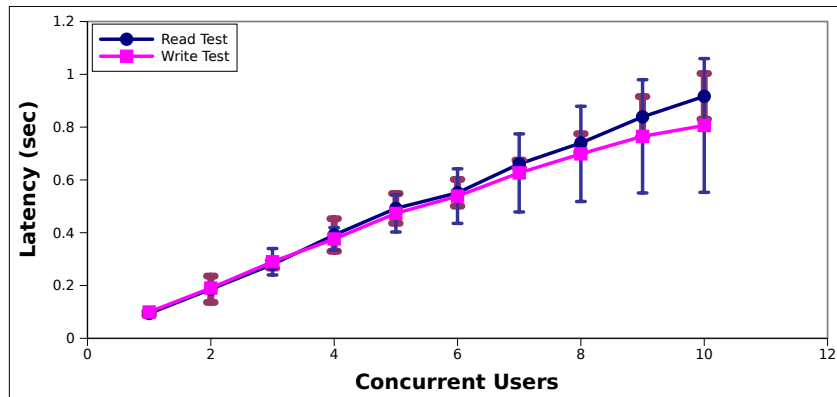


Figure 4.16: File latency of reading and writing files with respect to the number of concurrent accesses. For concurrent reads, the time to complete all reads scale approximately linearly. However, the variance also increases indicating that some reads must wait a longer time. Writes exhibit a similar pattern.

### Concurrent Access

FUSN filesystems also allow multiple users to concurrently access the filesystem. In order to minimize the time spent locking on the sensor node, locking is implemented on the host. Each sensor node interface has its own lock so that users accessing two different sensor nodes do not conflict. Currently, adaptive flow control mechanisms are not used (relevant for the CTP communications backend), although all read and write commands are processed by a sensor node before sending another command.

A concurrent read and write test was constructed to evaluate the effectiveness of the locking strategy. Both tests were executed on the host as Ruby scripts. For the read test, sensor nodes were loaded with the read filesystem containing a file with 10 values. Similarly, for the write test, the sensor nodes were equipped with a filesystem containing an empty file that appended values to that file. On the host, the number of concurrent threads that attempted to access the files was varied. As Figure 4.16 illustrates, the latency to complete a read and write operation increases linearly on average up to ten users. However, the standard deviation also increases with the number of users, indicating lock contention.

However, it should be noted that such a high number of concurrent accesses will be uncommon for most deployment scenarios.

### **4.1.5 Discussion**

Fully integrating computing resources as filesystem resources was initially explored in the Plan 9 operating system [80]. In Plan 9, all resources are mapped as files. For example, networking protocols, such as TCP/IP are mapped as a set of files that can be controlled using standard read and write operations. Although originally designed for standard distributed systems, Tilak [96] and Pisupati et al. [81] explored directly porting the Plan 9 protocols and interface to sensor networks. As a consequence, applications employ the filesystem interface as a programming model, with the limitations found in other message passing models.

FUSN, on the other hand, employs the hierarchical group model as the underlying programming model and uses Kensho to implement filesystem interfaces. These filesystems are end-user management and tasking interfaces. Users can interact with sensor networks using existing I/O libraries without understanding the complex protocols that drive the sensor network. Similarly, existing applications that interact with the filesystem, including many command-line programs, can use FUSN filesystems to manage and interact with the sensor network.

Although FUSN filesystems are very different from Tables and the hierarchical group model, FUSN can still take advantage of the hierarchical group model to simplify implementation. Several local tasks are assigned to all sensor nodes. These tasks respond to various FUSN commands. Communication is handled via the Kensho “push”, “publish”, and “collect” functions. Although FUSN does not employ collective tasks, the FUSN filesystem server can still access data published by the local tasks. FUSN does not employ logical groups (outside the primary global group). Although users can construct groups

of sensor nodes (by creating directories and moving sensor nodes into these directories), these groups are primarily organizational and not necessarily functional. As such, group membership information resides completely on the filesystem server.

FUSN, like Tables, also requires node-specific communication. In FUSN, sensor nodes are represented as directories that contain various files. As such, FUSN must send messages and commands to specific nodes. Although the hierarchical group model discourages node-specific communication, this was necessary in order to preserve filesystem semantics. Even with additional node-specific communication requirements, FUSN demonstrates that the hierarchical group model is useful for applications with very different interaction models.

## **4.2 Privacy Applications**

Many sensor networks in social settings will require new privacy and confidentiality guarantees in order to protect individual participants [92, 95, 86]. Unfortunately, privacy and confidentiality issues have not been adequately addressed in sensor networks for at least two reasons. First, many applications, such as environmental data monitoring, may not have strong privacy requirements. Second, implementing traditional forms of data protection on sensor nodes requires relatively complex communication and computation. For example, although recent research has demonstrated that encryption is possible on sensor nodes [79, 100], the relative overhead remains imposing. Key distribution remains challenging [36] and may require a complex message passing protocol.

In order to address these issues, new privacy-preserving algorithms for sensor networks are necessary. These algorithms, besides protecting data, should also conform to the hierarchical group model. This ensures that the overall design and implementation of the algorithm can be integrated into other existing applications. Two algorithms that address

two broad classes of applications are presented. The first algorithm, the negative survey, is designed for anonymous data collection from a large number of sensor nodes (primarily in urban scenarios). The data is used to create a histogram, which is then reported back to the sensor nodes. The second algorithm focuses on individual privacy and is designed to protect location data (usually collected via GPS). Although designed for different scales and applications, both algorithms take advantage of *negative* representations of data. By doing so, the algorithms conform to the communication and computational patterns described by the hierarchical group model, simplifying implementation.

### 4.3 Negative Survey

The negative survey consists of a set of protocols that enable anonymous data collection [56]. These protocols and the discussion regarding their information-theoretic characteristics were originally detailed in [37] and are replicated here for completeness. Confidential information is protected by ensuring that sensor nodes, instead of transmitting their actual data, transmit a data value that was *not* collected. The basestation then uses these negative samples to reconstruct a histogram of the actual data. These protocols are computationally simple and do not increase communication overhead relative to the original application.

#### 4.3.1 Selecting a Negative Category

Every sensor node chooses from the same set of categories, and independently determines what data to transmit back to the basestation. The algorithm used by the sensor node selects data from a finite set of discrete data values. For many applications, such as traffic monitoring, these data values represent mutually exclusive and exhaustive categories. For example, categories for traffic monitoring would consist of speed increments (0 – 9 mph, 10 – 20 mph, etc). When discussing the node protocol, the terms *categories* and *data*

*values* are used interchangeably.

Each node in the sensor network occasionally receives a *query* requesting data. Upon receiving a query, the node first identifies the initial category  $p$ . Instead of transmitting  $p$  to the basestation, the node selects another category uniformly at random and transmits this category. More precisely, let  $U$  be the set of all categories. The node then chooses a category uniformly at random from the set  $U - \{p\}$ . In this way, nodes are said to transmit *negative* values. If the sensor node transmits  $p$ , the original category, the node is said to be participating in a *positive survey*.

If an adversary intercepts a transmission from a sensor node, he or she learns only a category that the sensor node did not record. Assuming that there are more than two categories, the protocol preserves a high degree of privacy by making it difficult to correctly guess the actual category which was sensed. The node protocol is computationally simple and does not increase communication overhead because the number of messages transmitted remains the same compared to a positive survey.

### 4.3.2 Reconstructing the Histogram

Once the sensor nodes transmit the negative values, the basestation reconstructs the original frequency distribution. In order to do this, the basestation must know both the number of sensor nodes and the set of categories used by the nodes. Given  $t$  categories and  $n$  sensor nodes, the basestation receives  $R_i$  replies for category  $i$  from the sensor network. The basestation then estimates  $A_i$ , the actual number of nodes that belong in category  $i$ .

In order to calculate  $A_i$  for all  $i$ , the basestation uses the equation:

$$A_i = \sum_{j \neq i} (R_j - \sum_{k \neq i, j} C_{j,k})$$

where  $C_{i,j}$  is the expected number of sensor nodes in category  $j$  that report  $i$ .



Chapter 4. Applications of the Hierarchical Group Model

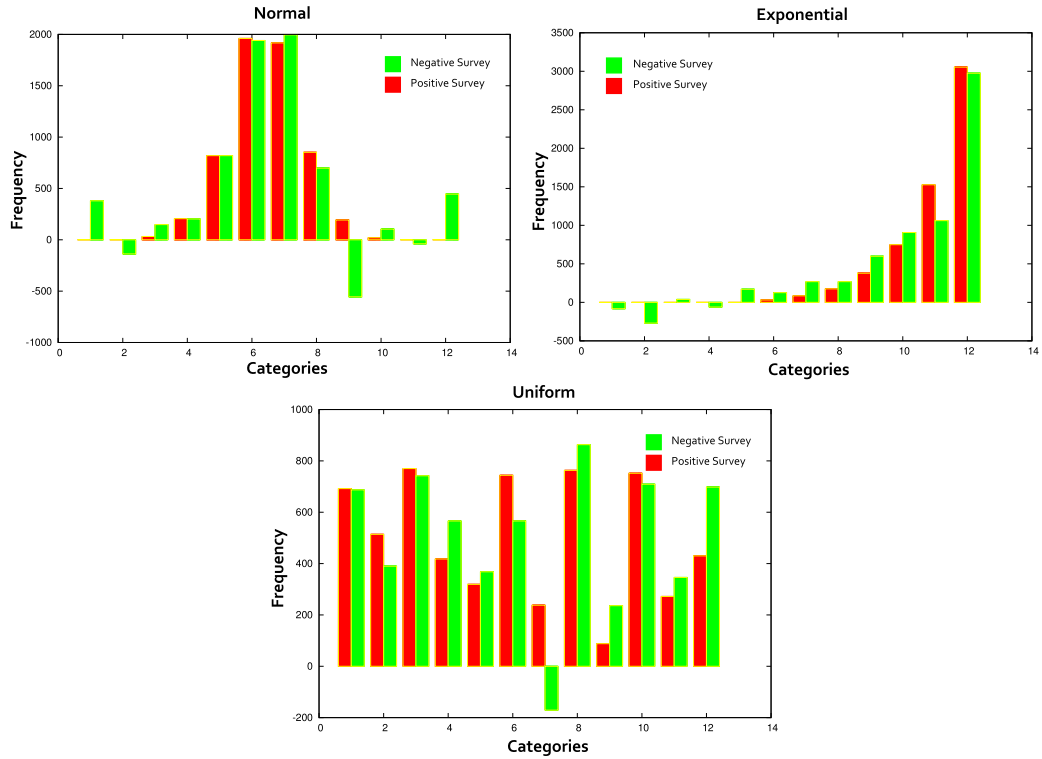


Figure 4.17: Reconstructed histograms with the corresponding actual histogram for three distributions. Each trial used twelve categories and 6000 sensor nodes. The negative survey is able to capture the general shape of the histograms for all three distributions.

Since each sensor node transmits one of the other categories, the probability of selecting another category is  $\frac{1}{(t-1)}$ . Using this, the equation is simplified to

$$A_i = n - R_i(t - 1).$$

Using this function, the basestation is able to reconstruct the histogram of original values by iterating over all  $i$  categories.

### 4.3.3 Implementation and Evaluation

The negative survey has been implemented in both a simulation environment (using *Matlab*) and on the TelosB sensor platform using the Kensho programming interface. The simulation implementation was used in order to evaluate the accuracy and applicability of the negative survey reconstruction method. This was necessary since the number of sensor nodes used in the negative survey tests was very large.

Several parameters, including the number of sensor nodes, the number of categories used in the survey, and the distribution of the data (positive) values, were varied and tested to evaluate the conditions under which the negative survey performed well. Sensor nodes were restricted to choose only a single category per query. This restriction is an example of the tradeoff between protecting the confidentiality of a node's data values and the ability of the basestation to reconstruct the data. This tradeoff can be managed for particular applications by varying the number of sensor nodes participating in the survey and varying the number of categories each sensor node transmits.

Reconstruction tests were evaluated against pre-selected distributions; each node was assigned a random variable drawn from that distribution, which indicated its *positive* category. The simulation ran the node protocol on each sensor, transmitted the negative data, and then ran the basestation protocol to reconstruct the distribution. Three different distributions were used: normal, exponential, and uniform. The uniform distribution chooses each category with uniform probability between 0 and 1. Each test was run with twelve categories and 6000 sensor nodes. The results were normalized and compared against the original distribution using the following root mean-square error (RMSE) test:

$$RMSE = \sqrt{\sum_{i=1}^n (positive(i) - negative(i))^2}$$

As Figure 4.17 illustrates, the reconstructed histogram matches the original distribution well for all three distributions. However, the negative survey occasionally generates

## Chapter 4. Applications of the Hierarchical Group Model

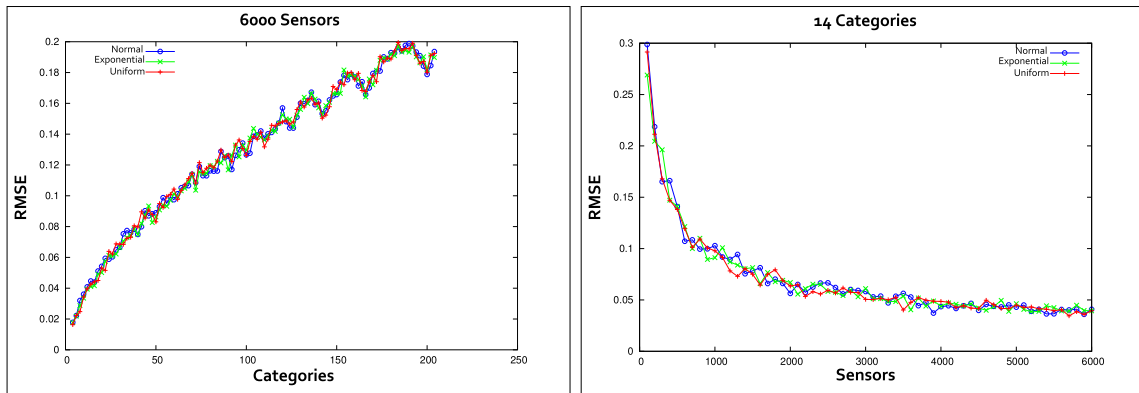


Figure 4.18: Error in the reconstructed histogram with respect to the number of categories and samples. The error increases with the number of categories. As the number of sensor nodes is increased, the error initially decreases quickly and subsequently decreases at a slower rate.

frequencies less than zero. These values arise when the expected contribution for a particular category exceeds the actual reported total for that category. This is a statistical artifact; as the number of samples is increased the number of frequencies less than zero will decrease.

### Varying the Number of Categories and Samples

In order to understand the effects of the number of categories and samples on error, tests were conducted that varied the number of sensor nodes participating in the survey, and the number of categories from which each sensor node must choose. Intuitively, the error is expected to decrease as the number of samples increases, assuming a constant number of categories. In addition, the error is expected to increase as the number of categories is increased since the number of choices for the sensor node also increases.

Six thousand sensor nodes were used for the first test. The number of categories was varied from 4 to 204 in increments of 2. This test was run independently ten times and results were averaged from all ten runs. This test was run using the normal, uniform,

Chapter 4. Applications of the Hierarchical Group Model

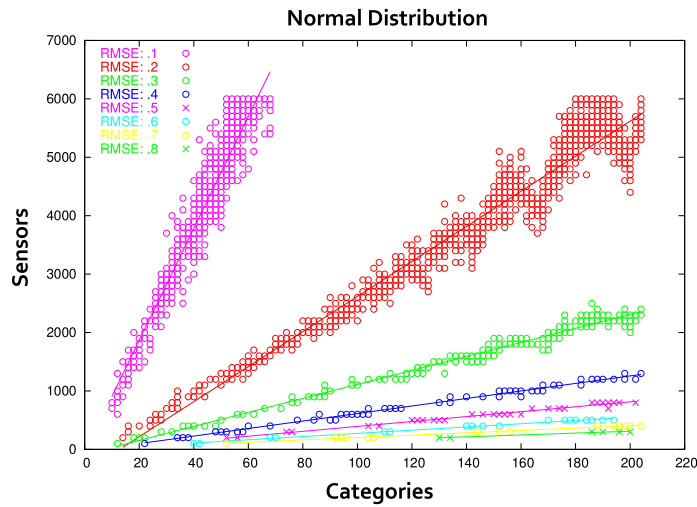


Figure 4.19: As the number of categories is increased, the number of sensor nodes needed to maintain a constant RMSE also increases. Each point represents a RMSE value within a  $\pm.008$  range of the desired RMSE value.

and exponential distributions. Figure 4.18 shows that error increases with the number of categories in a near-linear fashion for all three distributions.

For the second test, the reconstruction accuracy was compared against the number of sensor nodes. Sensor nodes were simulated to choose from 14 categories while the number of nodes varied from 100 to 6000 in increments of 100. Again, this test was run using the normal, uniform, and exponential distributions. The resultant error was averaged from ten independent runs. Figure 4.18 shows that the error falls off quickly as the number of samples is increased and then levels off.

In order to compare the error associated with the negative survey to a baseline sampling error, a simple positive survey with 14 categories was tested where the number of sensor nodes was varied from 100 to 6000. RMSE was used to characterize the difference between the positive survey histogram and the actual histogram. As expected, the error quickly decreases to zero for both the normal and exponential distributions as the number of sensor nodes was increased (data not shown). Although the error associated with the uniform

## *Chapter 4. Applications of the Hierarchical Group Model*

distribution also decreases, it does not reach zero due to the nature of the distribution.

Given a target RMSE, the relation between the number of categories and the number of sensor nodes was tested. This information is useful for applications where a tolerable error threshold is known beforehand. In Figure 4.19, the number of sensor nodes needed to maintain a target RMSE is plotted while increasing the number of categories. This was done for target RMSE values between 0.1 – 0.8 in increments of 0.1. As the number of categories increases, the number of samples to maintain a desirable RMSE value must also increase. All three distributions (normal, exponential, and uniform) behave similarly. This implies that an application can use one method for maintaining a constant accuracy without necessarily knowing the distribution of the data ahead of time.

### **4.3.4 Applications of the Negative Survey**

The negative survey is appropriate for applications in which the distribution of data is important rather than specific answers from sensor nodes. For example, an application in which users want to know how busy a restaurant is could aggregate discrete location data (such as a city block) provided by individual users' mobile phones.

Another potential sensor network application requiring anonymization is automobile traffic monitoring [35, 57]. Traffic monitoring is used in major cities, for example, to make decisions regarding street layouts. It can also be used to identify bottlenecks due to traffic signals. Traffic monitoring could also be useful for individuals. Some road intersections may be congested, while others may be frequented by dangerous drivers. By monitoring traffic conditions, individual drivers could avoid roads with problematic conditions.

Although aggregated information about traffic could be useful, both for individuals and traffic engineers, most drivers would naturally be reluctant to have their driving monitored for fear of legal or insurance repercussions. If, however, the privacy of individuals could be guaranteed, then the larger community could benefit from aggregated information without

#### *Chapter 4. Applications of the Hierarchical Group Model*

loss of individual privacy. Similar considerations constrain the collection of health information in epidemiological settings. Although privacy enhancing databases address some concerns, they generally require the individual to trust that his or her information will be sanitized in a way that protects privacy.

In the traffic monitoring example, the negative survey is used to provide end-to-end anonymity to individual drivers. Observers monitoring the traffic would still have access to the real traffic distribution. Assuming that each vehicle is equipped with a speed sensor, the speed sensor records the current speed of the host vehicle and the actual speed limit of the road on which the vehicle is traveling (provided possibly by a basestation located near the road). The basestation collects the sensor data and performs the histogram reconstruction within a locally constrained area, e.g., a single intersection or section of roadway. Each sensor contains a list of pre-determined categories. For this application, each category represents a set of relative speeds above and below the speed limit. An example with six categories is given below:

1. 10+ mph over the speed limit
2. 5 - 9 mph over the speed limit
3. 0 - 4 mph over the speed limit
4. 0 - 4 mph under the speed limit
5. 5 - 9 mph under the speed limit
6. 10+ mph under the speed limit

In order to determine the proper category, each sensor node takes the difference between its current speed and the known speed limit and chooses a category according to the node protocol. The sensor node then transmits this negative value to the basestation.

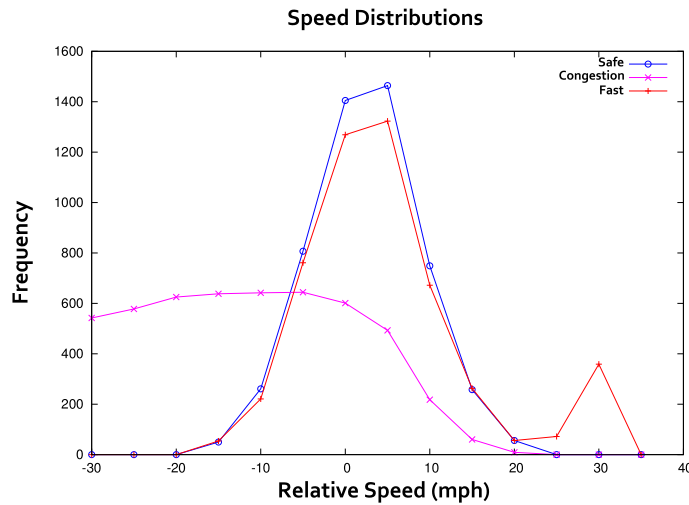


Figure 4.20: Three speed distributions to characterize different traffic conditions. Normal traffic is represented by a normal distribution. Traffic in which a few vehicles travel faster than most is represented by a bi-modal distribution. Congested traffic is represented an approximate uniform distribution with a long tail.

The basestation, in turn, receives data from all the sensors and reconstructs the histogram. After constructing the histogram, the basestation classifies the histogram into one of three traffic behaviors. Each traffic behavior is distinguished by a canonical speed distribution (illustrated in Figure 4.20). These speed distributions attempt to capture *congested*, *safe*, and *fast* traffic behaviors. For this application all traffic is assumed to obey one of these three behaviors.

A *safe* speed distribution is characterized by a normal curve centered in the  $0 - 4$  mph category. A *congested* speed distribution is characterized by a skewed-normal curve that leans towards the categories under the speed limit. Finally, the *fast* distribution is a bi-modal curve. The larger mode represents speeds centered near the  $0 - 4$  mph category, while the smaller mode is centered near the faster speeds. These speed distributions were derived from real-world patterns [32].

The simulation recorded the average classification accuracy with respect to the num-

## Chapter 4. Applications of the Hierarchical Group Model

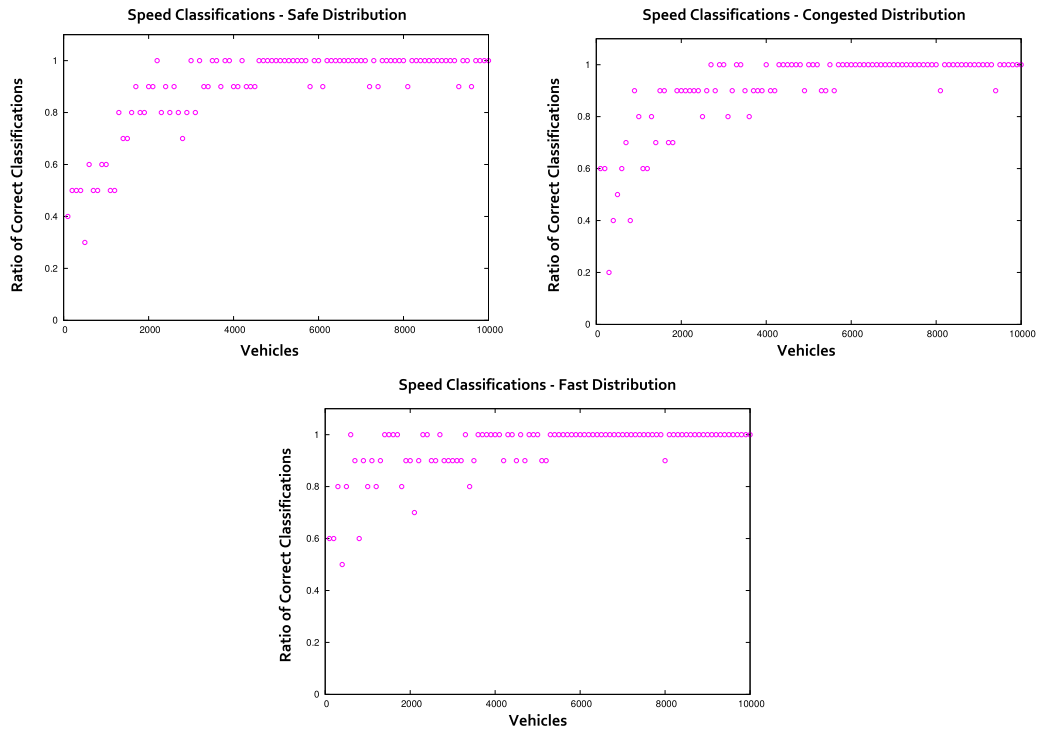


Figure 4.21: Classification accuracy for three traffic conditions. Each point is the percent of correct classifications over ten trials. The classification performs well for all three traffic distributions with over 3000 samples.

ber of vehicles participating in the survey. The accuracy was measured as the ratio of the number of correct classifications to the total number of classifications over ten independent runs. The number of vehicles was varied from 100 to 10000 in increments of 100. Each sensor had access to 12 speed categories similar to the example with 6 categories. The experiment was run once for each of the speed distributions. Within a single experiment, the actual speed distribution remained constant. After the basestation constructed the histogram, the negative survey results were compared to the three canonical speed distributions using a modified RMSE test. The comparison that yielded the lowest RMSE value was chosen as the actual speed distribution. Results of this test are shown in Figure 4.21. In order to validate the algorithm, the same test was conducted using a positive survey protocol. 100% accuracy was observed using the positive histogram for all settings.



## *Chapter 4. Applications of the Hierarchical Group Model*

Therefore, any error is due to the inaccuracy of the reconstructed histogram.

The classification scheme performed well for all three speed distributions. On average, classification accuracy reaches 80% with 3000 readings. Increasing the number of vehicles increased accuracy to over 90% and eventually to 100%. More complex classification algorithms could increase the accuracy (or the number of categories could be reduced) to improve accuracy in settings with a low number of vehicles. However, 4000 – 6000 vehicles in a traffic area is consistent with typical highway and interstate flow<sup>1</sup>. These results illustrate the kind of data rates that would be appropriate for a negative survey approach.

The purpose of this application scenario is not to demonstrate a real-world classification algorithm for traffic monitoring. Real-world deployments would likely include more kinds of traffic behaviors and use more sophisticated classification algorithms. The application does show, however, that a negative survey could supplement existing applications to increase anonymity.

### **4.3.5 Discussion**

Urban applications that present aggregate information have the potential to be widely useful. However these applications risk revealing private information while performing the aggregation. In order to address this issue, the negative survey employs a simple, but effective mechanism to anonymize confidential information. Sensor nodes, instead of transmitting the raw sensor data, transmit the complement of the data. The basestation, using these negated values, constructs a histogram of the original data. Other issues, such as security against malicious adversaries, are not explicitly addressed by the negative survey and are dealt with complementary techniques [19, 83].

Another set of algorithms with similar goals to the negative survey are the data pertur-

---

<sup>1</sup>[http://www.mrcog-nm.gov/maps\\_on-line.htm](http://www.mrcog-nm.gov/maps_on-line.htm)

bation algorithms proposed by Agrawal *et al*[12] and more recently by Zhang *et al*[107]. Their technique perturbs the original data set with random noise drawn from a known distribution. This perturbed data is then used to reconstruct the original distribution (often using Bayes Theorem).

In contrast to the negative survey, the data perturbation algorithms assume that the data and the additive noise are drawn from a continuous domain. The negative survey assumes the opposite: the data and locations are drawn from a set of discrete values. Although raw sensor data approximates continuous data, many applications classify sensor data into a few discrete categories (such as walking, running, driving, etc) [74]. By integrating with such applications, the negative survey can provide anonymity to these classifications.

Because the sensor nodes can perform the anonymization step independently, all computation and communication can be sufficiently described using the hierarchical group model. The local protocol is implemented as a Kensho local task, while the basestation protocol is implemented as a collective task. Commands and anonymized data are implemented using the standard Kensho communication functions. Since the negative survey is described so well by the hierarchical group model, other applications written in Kensho that require aggregate anonymization will be able to integrate the negative survey with little additional work.

## 4.4 Location Anonymity

As more devices become integrated with location sensors, location-aware applications will become ubiquitous. Location services will be provided by many different types of sensors including GPS, WiFi triangulation, and cellular tower identification. Applications include cars that provide accurate driving directions, phones that notify users of nearby friends [7], and cameras that automatically “geotag” the location of a picture. Although useful, these

## Chapter 4. Applications of the Hierarchical Group Model

applications have a privacy cost. For example, applications designed to help locate friends could be used to track users without their knowledge.

Similar technology can be used for more intrusive forms of monitoring. For example, several states have recently passed a version of “Jessica’s Law” [6], including California’s Proposition 83, that mandates that released or paroled sex offenders wear ankle bracelets equipped with GPS units. Similar proposals have been made for monitoring the locations of taxicabs in large cities. The GPS monitors record the location of the parolee periodically. These records can then later be correlated against a set of known crime locations by a parole officer. In the event that the parolee is near a suspect area, the correlation can be used evidence of wrong-doing. This example provides more information to the parole officer (every location the person visits) than what is needed to determine if the person was near the scene of a crime or to determine if a banned location was visited. Although we may not have much sympathy for protecting the privacy of a convicted sex offender, it is easy to imagine extensions to workplace monitoring or to parental monitoring of children.

Like the negative survey, the approach taken for these problems is simply *not* to store or transmit the actual location, but to use a representation that allows the correlation between different locations to be computed without compromising the actual location. Since the devices only store the negative representation of the location, the possibility of compromising private information is minimized.

### 4.4.1 Anonymized Locations

The anonymization algorithm allows a user Alice’s location to remain hidden, while ensuring that a query of the form “Was Alice at Location X at Time Y?” can be answered correctly. The query will return the correct answer with high probability even if Alice is not exactly at the location but close. The algorithm accomplishes this by processing and storing location data using an encoding in which geographically close locations are

```
Storing Location algorithm:  
  
INPUT: location  $l$   
OUTPUT: Negative database,  $NDB$ , that stores  $l$   
  
0.  $q = \text{quadrant\_encoding}(l)$   
1.  $q' = \text{obfuscate}(q)$   
2.  $NDB = \text{new\_singleton}(q')$   
  
Querying Location algorithm:  
  
INPUT: location  $l$  and negative database  $NDB$   
OUTPUT: Boolean,  $m$ , indicating whether  $l$  is  
contained  $NDB$   
  
0.  $q = \text{quadrant\_encoding}(l)$   
1.  $q' = \text{obfuscate}(q)$   
2.  $m = \text{check\_membership}(q', NDB)$ 
```

Figure 4.22: Overview of the encoding and querying algorithm. The encoding process takes in a location value and outputs a negative database containing a randomized representation of the location.

close in Hamming Distance and geographically distant locations are not. The algorithm requires several steps, outlined in Figure 4.22. Given a location  $l$ , the algorithm first maps the location to the quadrant encoding to achieve the Hamming Distance property; it then randomizes  $l$  in a special way that preserves the Hamming Distance property. This is to ensure that  $l$  cannot be easily obtained by an adversary. Finally, the randomized value is stored in a singleton negative database, so that queries can be answered correctly even if the exact location isn't presented (fuzzy matching). After creating the negative database, the original location value is discarded.

### Quadrant Encoding

The system starts with a GPS value in the standard US government data format [4]. Since the algorithm focuses on latitude and longitude queries, additional information such as altitude are discarded. Afterwards, the latitude and longitude are mapped to a quadrant encoding similar to a Quad Tree [41]. In the quadrant encoding, the total area is divided into four equal quadrants. Each of these quadrants are in turn recursively subdivided into

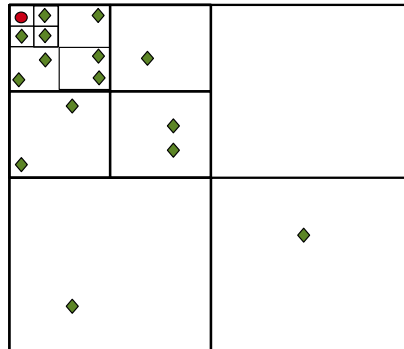


Figure 4.23: The geographic quadrant encoding scheme. The geographic area is recursively subdivided into four quadrants. During each recursion step, the location is placed one of the four quadrants.

higher resolution quadrants (Figure 4.23). This recursion is continued until the maximum resolution of the location device is reached (approximately 3 meters for GPS). At each recursion step (referred to as a quadrant level), the location is placed within one of the quadrants. The quadrant number is then recorded to encode the location. In the United States, it takes approximately 18 quadrant levels to cover a typical state. Smaller states take approximately 16 quadrant levels. Typical cities take approximately 14 quadrant levels, while smaller cities takes less than 13 quadrant levels.

### Binary Representation

After converting a location to a list of quadrant values, the quadrant values are converted to a binary representation. In contrast to a naïve encoding, the binary encoding must have the property that two geographically close locations encode to two binary strings with low Hamming Distance (Hamming Distance measures the number of differing bits). This ensures that two nearby locations have a higher probability of matching in a negative database.

Chapter 4. Applications of the Hierarchical Group Model

Quadrant	Low Level	High Level
0	0 0	0 0 0 0 0 0 0 0 0 0 0 0
1	1 0	1 1 1 1 1 1 1 1 0 0 0 0
2	0 1	1 1 1 1 0 0 0 0 1 1 1 1
3	1 1	0 0 0 0 1 1 1 1 1 1 1 1

Table 4.5: Examples of how quadrants in two extreme quadrant levels are encoded in binary. The binary representations of the lowest quadrants have a Hamming Distance of 1 between any pair of quadrants. The highest quadrants have a Hamming Distance of 8.

This property is ensured by making the following observation: two geographically close locations will have a nearly identical list of quadrant values. Since the quadrant encoding algorithm is recursive, two nearby locations will share the same quadrant values for the higher quadrant levels (i.e. the same city), but differ for the lower quadrant levels (i.e. different neighborhoods). This observation is put to use in our encoding scheme such that lower quadrant levels are encoded differently than higher quadrant levels. In a lower quadrant level, the binary values of the four quadrants are constructed so that the Hamming Distance between any pair of quadrants is 1. Similarly, for higher quadrant levels, the quadrant binary values are constructed with a Hamming Distance of 8. Figure 4.5 illustrates the binary values of each of the four quadrants in both the lower and higher quadrant levels.

For the purposes of evaluating the system, the lowest three quadrant levels were designated (corresponding to a resolution of approximately 12 meters) as the lower quadrant levels, while the rest were designated as the higher quadrant levels. Using this binary encoding, two nearby locations will have a *maximum* Hamming Distance of 3 (since they can only differ in the first three quadrant levels). However, two far locations will have a *minimum* Hamming Distance of 8 (since they must differ in at least one of the higher quadrant levels).

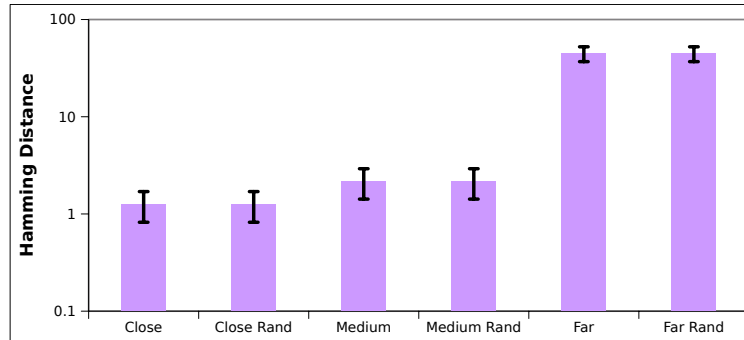


Figure 4.24: Hamming Distance properties after randomization. Locations close to the stored location are close in Hamming Distance, both before and after the randomization step. Columns labeled *rand* in the figure correspond to the randomized representation. *Close* locations vary only in the lowest level quadrant (approx. 3 meters); *Medium* locations vary in the second and third quadrant levels; and *Far* locations vary in the highest level quadrant.

Quadrants	HD Encoding	Randomized
2 2 0	01 01 000000000000	11 00 1101111110110
3 2 3	11 01 000011111111	01 01 100000101000
1 2 1	10 01 1111111110000	01 01 001010010000

Table 4.6: Location encoding in the quadrant representation. The *Quadrants* column shows the original recursive encoding, beginning with the highest level quadrant. The second column shows the *Hamming Distance* encoding, as described in the text, and the third column shows the *randomized* encoding described in the text.

### Randomization Step

Encoded locations are randomized before being stored on the device. This is done to make it difficult for adversaries to obtain the original locations. However, the randomization step must also ensure that the Hamming Distance properties of the encoded locations are preserved.

The Hamming Distance property is preserved by randomizing two nearby locations using the same pseudo-random seed. Since two nearby locations will share a high-level

quadrant value, the algorithm uses the first high quadrant level value as the seed. Afterwards, the algorithm generates a pseudo-random binary string and XORs the encoded location with that string. Since two nearby locations use the same random seed, they will generate the same pseudo-random binary string. As such, the results of the XOR operation on the two nearby locations will only differ in the places where the original encoded locations differed, thus preserving the Hamming Distance property.

As Figure 4.24 illustrates, the randomization scheme successfully preserves the Hamming Distance property. The Hamming Distance between two close locations is approximately 1 before and after randomization. The Hamming Distance between two distant locations, however, is nearly two orders of magnitude more before and after randomization. Figure 4.6 illustrates a complete example of quadrant values after being encoded and randomized.

### Singleton Negative Database

After the previous two steps, the algorithm produced a string that represents a single geographic location using the quadrant encoding and subsequent randomization. The next step is to support fuzzy retrievals, by which a query such as “Was Alice at Location X?” will with high probability return a positive answer, if Alice is geographically close to X. This is accomplished using a data structure known as a negative database [39, 38].

Negative databases, unlike normal (positive) databases, store a compact representation of the complement of the data. Specifically, given a binary record  $s$  chosen from some finite set  $U$  ( $U$  is the universe of all possible records), negative databases store a set of records that form a subset of  $U - s$ . This subset is compressed by expressing the negative database over the alphabet  $\{0, 1, *\}$  where  $*$  is known as the “don’t care” symbol, that matches either a 0 or 1 symbol in a given bit position.

Esponda et al. show how to construct negative databases that are hard-to-reverse [39,



```

INPUT:  A binary string  $s$ 
        Integers  $L$  and  $k$ 
        Floating point numbers  $0 < q < 1$  and
         $r > 0$ 

OUTPUT: A negative database,  $NDB$ , that does
        not match  $s$ 

0. Let  $n \leftarrow L * r$ , initialize  $NDB = \{\}$ 
1. Repeat
2.   Select  $k$  distinct positions,  $\gamma$ ,
   uniformly at random from  $[0, L - 1]$ 
3.   Create a string  $z$  of length  $L$  and set
    $z[\gamma] = s[\gamma]$ 
   Set the remaining positions to *, the
   "don't-care" symbol
4.   Repeat
5.     For each position  $i$  in  $\gamma$ 
6.       Complement the value of  $z[i]$ 
       with probability  $q$ 
7.     Until at least one value has been
       changed
8.   Add  $z$  to  $NDB$ 
9. Until  $|NDB| \geq n$ 
    
```

Figure 4.25: Overview of the singleton negative database algorithm. The singleton algorithm accepts a binary string and creates a negative database containing that binary string.

38], i.e. given a negative database,  $NDB$ , as input, it is computationally intractable to infer the original (positive) data. In this work, negative databases created using the algorithm are relatively easy-to-reverse. This algorithm, informally known as the singleton algorithm

<i>Original Record</i>	<i>NDB</i>
0000110010110000	* * 1 * * * * 0 * * * * 0 * * * *
	* * 0 * * * * 1 0 * * * * * * * *
	* * * * * * * * 1 1 * * * * 0 * *
	* * * * 0 * * 1 * * 0 * * * * * *
	1 * * * * * 1 * 0 * * * * * * * *

Table 4.7: Example of a singleton negative database. The negative database consists of multiple entries (records). \* characters are “don’t cares” that match either a 1 or 0. The database is constructed such that the *Original Record* fails to match any of the entries in the negative database, and the only other records that fail to match are guaranteed to be within close Hamming Distance. Parameters of the algorithm determine the probability of these additional records occurring and what their Hamming Distance will be.

## Chapter 4. Applications of the Hierarchical Group Model

(Figure 4.25), generates a negative database by iteratively creating an entry that does not match the original record in at least one bit position. The algorithm depends on a user supplied parameter  $k$  that specifies the number of explicit bits in each  $NDB$  entry. The algorithm then chooses  $k$  random locations within the record, randomly flips those location values, and fills the rest of the entry with  $*$  symbols. The total number of entries is a product of a “difficulty” parameter  $r$  and the length of the original record  $l$ . An example of a negative database with five entries is shown in Figure 4.7.

### Fuzzy Location Querying

After storing the location in a negative database, the original location is discarded. After this step, a user can query the negative database to determine if a particular location is stored in the negative database. In order to query for a particular location, the user must first convert the desired location using the encoding and obfuscation scheme described earlier. Afterwards, the encoded location,  $s$ , is compared to the negative database.

The singleton algorithm creates multiple record entries (superfluous strings) within a single  $NDB$ , and all the extra entries are guaranteed to be within close Hamming Distance of the original (singleton) record. The definition of *close* is governed by the parameters  $r$  and  $q$ . An interplay between  $q$  and  $r$  determines how difficult the  $NDB$  will be to reverse and how many superfluous strings will go unmatched by  $NDB$ .

Location  $s$  is said to be in  $NDB$  if  $s$  matches at least one of the entries. A match between  $s$  and an entry is determined bitwise by checking that the value at each bit position is either identical or a  $*$ . Similarly, since  $NDB$  represents the entries *not* in  $DB$ ,  $s$  can be said to be in the original database if it doesn't match *any* of the entries in  $NDB$ . Because each  $NDB$  encodes a single location, the system must create a separate  $NDB$  for each timestamped location. Thus, querying for a particular location is linear in the size of the  $NDB$  and the total number of timestamped locations.

## Chapter 4. Applications of the Hierarchical Group Model

The singleton *NDB* has the property that an encoded location  $s$  with a low Hamming Distance (relative to the location stored in *NDB*) will have a higher probability of not matching any entry in the negative database. This property, known as fuzzy matching, was first described by Jia [59]. Combined with the Hamming Distance property described for the quadrant encoding, fuzzy matching ensures that two geographically close locations will have a high probability of matching compared to locations that are farther apart.

### 4.4.2 Evaluation

The performance of the method was first studied by examining its accuracy at storing and querying for specific locations. The difficulty of “reversing” the actual location stored in the negative database was also examined. In both cases, a simulation was used to determine that the approach is useful under a wide range of parameter values while remaining difficult to break.

The algorithm contains several parameters:  $r$  (related to reversal difficulty),  $k$  (the number of specified bits in a *NDB* entry), and  $q$  (the number of quadrant levels encoded by each location). For all tests performed  $k = 3$  and  $q = 16$ . Because  $r$  affects both the size of the negative database and the expected number of fuzzy matches,  $r$  was varied across a range of values.

#### Accuracy

Because  $r$  controls the number of entries in the *NDB*, the probability of matching one of the *NDB* entries given an input location increases with  $r$ . To measure this effect, a random location was generated, encoded, and stored in an *NDB*. Three additional test locations were generated to query against the *NDB*. The first test location, *close*, was generated to differ only in the lowest quadrant level (and hence shares all other quadrant levels). This

Chapter 4. Applications of the Hierarchical Group Model

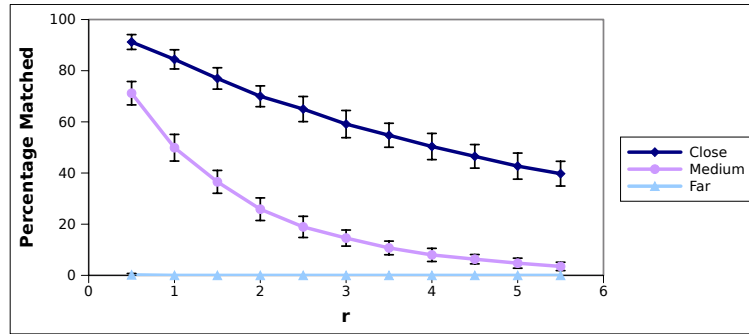


Figure 4.26: The number of matches with respect to  $r$ . Each plotted point is the number of matches over 100 runs for different values of  $r$ . The *close* values are for locations that differ in the lowest quadrant level to the stored location. The *medium* values differ in the second and third quadrant levels, while the *far* values differ in the ninth quadrant level. The number of matches for *close* and *medium* values are high for small values of  $r$  and decreases with  $r$ .

corresponds to a separation of approximately 3 meters. The second test location, *medium*, differed in the lowest three quadrant levels. Finally, the third test location, *far*, differed in one of the high quadrant levels (corresponding to approximately 768 meters).

Each test location was queried against the negative database. If the location did *not* match any entry, the result was labeled as a *match*. This test was performed for multiple

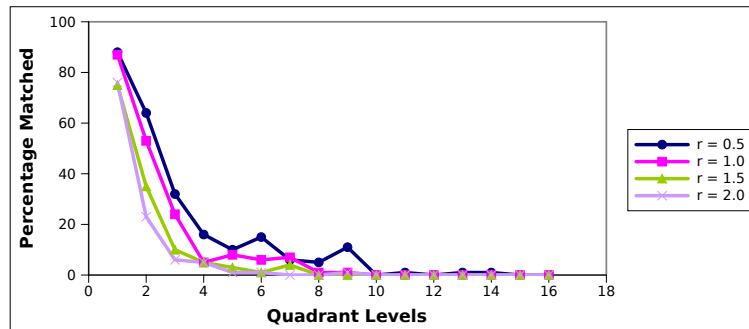


Figure 4.27: The number of matches as the geographic distance is increased with different values of  $r$ . Each plotted point is the number of matches for locations that differ in the indicated quadrant level. The number of matches quickly decreases after a short distance for all values of  $r$ . This measures the geographic resolution of the method.

#### Chapter 4. Applications of the Hierarchical Group Model

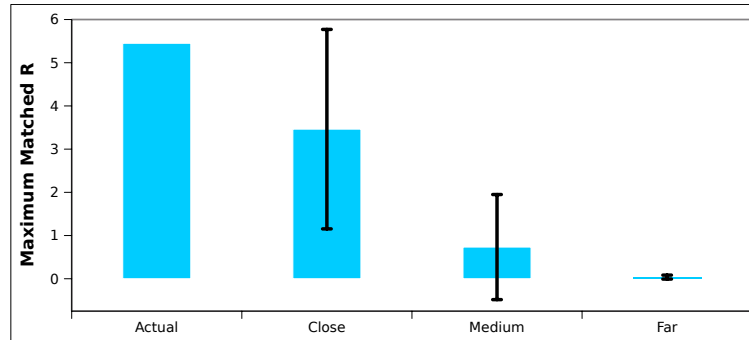


Figure 4.28: Maximum  $r$  value that generated matches for different distances. This result can be used to iteratively estimate the distance a guess is from the actual location.

values of  $r$  (0.5 to 5.5 in increments of 0.5). The total number of matches for each location over 100 trials was recorded. Since each trial involved generating a new set of random locations, the number of matches corresponds to the probability of a random location of a particular geographic distance matching the negative database.

As expected, the proportion of matches decreased as  $r$  was increased (Figure 4.26). This trend occurred for both close and medium locations. Close locations have nearly a 90% match rate when  $r$  is set to 0.5. However, the rate decreases to approximately 50% when  $r$  is increased to 5.5. A similar trend is observed for the medium distance locations. For far locations, there were no matches for any values of  $r$ , indicating those distances were simply too far to generate any false positives. This result shows that the  $r$  parameter can be used to control query resolution. For applications that demand highly accurate locations,  $r$  should be set to relatively high values. For applications that do not demand such accuracy,  $r$  could be set to smaller values.

The maximum  $r$  value that maintained a correct match for the close and medium locations were also examined. As Figure 4.28 illustrates, for close locations, most matches occurred when  $r$  was set to less than 3.5. Above  $r = 3.5$ , matches became unreliable. Similarly, for medium distance locations, most matches occurred when  $r$  was set to less than 1. For far distance locations, the number of matches was negligible regardless of the

## Chapter 4. Applications of the Hierarchical Group Model

$r$  value. Since the maximum number of *NDB* entries (determined by the  $r$  parameter) that match varies with distance, this result can be used to iteratively estimate the distance a particular guess is from the actual location.

The number of matches with respect to geographic distance was also compared while keeping  $r$  constant. Like the previous test, a random location was generated, encoded, and stored in a negative database. Test locations were generated to query against the negative database. The random test locations were generated so that each successive location was farther from the original record stored in the negative database (measured by the number of differing quadrant level values). This test was performed 100 times.

As Figure 4.27 illustrates, the number of matches is relatively high when the locations are close (differing only in the first and second quadrant levels). However, the number of matches drops quickly as the number of differing quadrant levels increases. This occurs for all values of  $r$  tested. This indicates that geographic distance plays a much larger role than  $r$  in determining the number of matches. This is in accordance with the original goals of the application since two far locations should not register a match.

### **Ease of Reconstructing the Original Data**

Besides storing location data accurately, the difficulty of reconstructing the original location stored in the negative database was assessed. The reversal difficulty has two components. The first is by storing the location in a negative database. Although the singleton algorithm was not optimized to maximize reversal difficulty, the negative database ensures that any successful reversal generates only an *approximate* stored location (because of superfluous strings). The second component is the randomization step described earlier, which prevents the adversary from obtaining the original location.

Although a precise characterization of reversal difficulty in this setting is left for future work, the effectiveness of these two components are quantified. First, the negative

Chapter 4. Applications of the Hierarchical Group Model

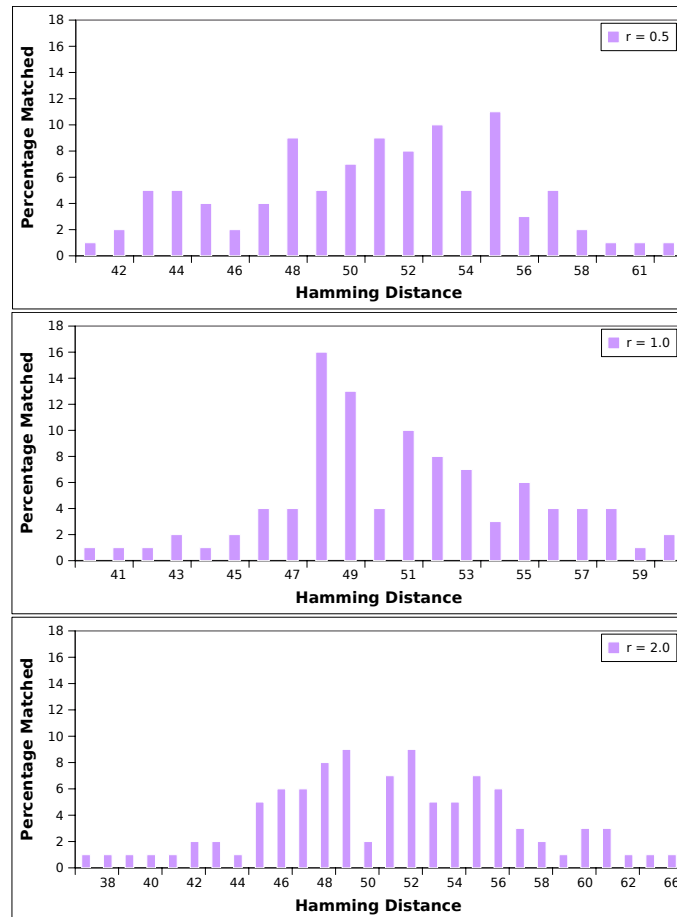


Figure 4.29: Frequency of solutions discovered by zChaff and their Hamming Distance from the original location. Most matches occur between a Hamming Distance of 45 and 55 indicating that the obtained solutions are not much better than random guessing.

database is reversed using zChaff, an efficient 3SAT solver [8, 73]. zChaff, although originally designed for 3SAT, can operate over negative databases by first converting the negative database representation to a 3SAT problem. Reversing the negative database corresponds to solving a particular 3SAT instance.

For this test a copy of the original location was retained before the randomization step. After reversing the negative database, the randomized location stored in the negative database was compared to the original location (measured by Hamming Distance). This

## Chapter 4. Applications of the Hierarchical Group Model

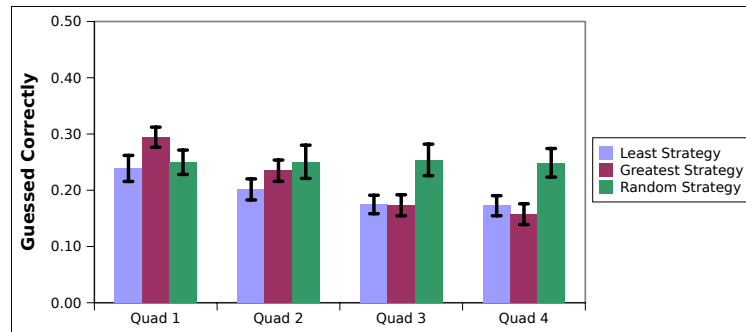


Figure 4.30: Number of correct guesses for three strategies that attempt to obtain the original location. Neither of the Hamming Distance strategies (least and greater) do better than the random strategy.

was done for 100 trials and across multiple values of  $r$  (Figure 4.29). For all values of  $r$ , locations estimated by zChaff had Hamming Distances between 35 and 60 with a mode near 45 to 55. This corresponds to a Hamming Distance of approximately half the number of total bits in the encoded location, indicating the estimated solution has a sufficient number of randomized bits.

Three different algorithms were used to evaluate the possibility of undoing the randomization step. The algorithms relied on the observation that for any given quadrant level, the encoding (before the randomization step) for each of the four quadrants is deterministic and known. The first algorithm, then, compared the randomized encoding for a given quadrant level with each of the four possible non-randomized encodings. The one closest to the randomized encoding (measured by Hamming Distance) was chosen as the quadrant. The second algorithm employed the same strategy, but instead chose the quadrant that had the highest Hamming Distance with the randomized quadrant level. Finally, a strategy that randomly selected one of the four quadrants was employed as a null model.

All three strategies were run 100 times. As Figure 4.30 illustrates, the randomization method is robust to both Hamming Distance strategies. Since there are four quadrants in each level, random guessing only guessed correctly 25% of the time. Neither strategy



performed better than random guessing for most quadrant levels.

### **4.4.3 Discussion**

Location-awareness is becoming an important feature in many social and safety applications. The features that make these applications useful, however, also increase the possibility of privacy violations. This section discussed a technique to encode and store locations such that the location data is hidden but can still be queried. Using a unique quadrant encoding scheme and a negative database, users can query for approximate locations without revealing all the locations a person has visited.

Both of the anonymization techniques discussed in this chapter employ local computation, hierarchical communication, and collective computation. The hierarchical group model was successfully applied since these anonymization techniques do not require any complicated key-exchange protocol. For the location anonymity technique, all sensor nodes were locally tasked with a GPS sampling function. This sampling function periodically collects, encodes, and stores a GPS value. In order to query for a location, the basestation issues a push with the relevant command to the sensor nodes.

Since the anonymization algorithms conform to the hierarchical group model, these algorithms can be easily integrated into other applications that employ the hierarchical group model (either via Kensho or Tables). Due to the simplicity of tasking functions in Kensho, existing applications can locally task these algorithms so that different logical groups can be made to share data anonymously. This demonstrates the flexibility of the hierarchical group model in accommodating sensor network oriented privacy algorithms.

# Chapter 5

## Related Work and Conclusion

The hierarchical group model is a way to abstract tasking and communication in sensor networks to simplify the construction of user applications. This simplification, however, does not unduly restrict the complexity of user applications. Using Kensho and Tables, users can define complex applications and algorithms. Additionally, by appropriately abstracting the process of tasking, the hierarchical group model lends itself to different implementation strategies. However, like all computational models, there are certain behaviors that are difficult to specify and implement. For the hierarchical group model and associated implementations of the model, computation that requires complex peer-to-peer coordination amongst the sensor nodes remains difficult to implement. However, it is important to note that such computation is not common for sensor networks. For applications that do require such coordination, other programming models must be used.

Currently there are two different instantiations of the hierarchical group model: Kensho and Tables. Both feature local and collective tasking, hierarchical communication, and logical groups. However, Kensho is a C-based system that uses threads as the underlying computational model. Tables, on the other hand, provides a data-driven functional model that interprets short functions whenever dependent data is created or changed. Al-

## *Chapter 5. Related Work and Conclusion*

though the systems are fairly different in their actual usage, the types of applications users can create on these systems are approximately the same since they share an underlying computational model.

For users not comfortable with a thread-based or data-driven approach, other underlying computational models, such as event-driven systems like TinyOS, can be adapted for use in the hierarchical group model. In that implementation, local tasks would consist of a set of event handlers that execute on the sensor node. Collective tasks, meanwhile, can be implemented as event-handlers that execute on more powerful basestations. These collective event-handlers would execute upon receiving messages from the local handlers. Communication between these tasks would obey the same hierarchical push / publish strategy. Finally, logical groups can be implemented using a strategy similar to the Tables implementation. By registering the admission function to the dependent data, the function can be activated as an event handler. The hierarchical group model affords many different underlying strategies. The choice between a thread-based or event-based implementation is largely determined by programming preferences. Fundamentally, these strategies combined with the hierarchical group model can implement the similar applications.

### **5.1 Related Work**

The material presented in this dissertation is related to previous work in computational and communication models. Several of these models have already been examined in this dissertation. Additional models used in other computational settings are examined in this section. Additional programming environments for sensor networks that include elements of different models and that focus on specific applications are also examined. Finally, different approaches to privacy and security are compared to the negative survey and negative database for their applicability in sensor network applications.

### 5.1.1 Computational and Communication Models

Many of the computational models explored in sensor networks have an origin in high performance and distributed computing. Typically these computational environments have very different goals from sensor networks. High performance applications are often used to simulate some computationally demanding process (atomic reactions, biological processes, etc). Consequently early work on computational models focused on maximizing performance. One of the most popular communication models is the Message Passing Interface (MPI) [45]. MPI, like the hierarchical group model, provides support for a variety of different communication patterns, including collectives and point-to-point communication. This communication takes place within MPI groups. However, unlike the hierarchical group model, MPI groups are static and describe processes instead of nodes. In that way, MPI is similar to other sensor network messaging models such as the TinyOS active message model.

The recent trend towards massively multi-core architectures, has led to models that also attempt to simplify programming in large-scale systems. These models, like the sensor network models, also abstract communication patterns [28] and provide a whole-system view [47]. This problem is made more difficult since, unlike sensor networks, communication and computation patterns are difficult to generalize across most high performance applications [24].

Applications with regular communication patterns, however, can take advantage of MapReduce [31], a restrictive programming model developed for large-scale data processing at Google <sup>1</sup>. In MapReduce, user defined *map* functions operate over key-value pairs and generate intermediate data pairs. The system automatically partitions the data and maps computation over a set of physical compute nodes. Similarly, user defined *reduce* functions operate over a list of data and produce a shorter list of data (often a single

---

<sup>1</sup>[www.google.com](http://www.google.com)

value). This programming model is similar to the hierarchical group model in that functions are defined without explicitly specifying where these functions execute. However, the semantics of local and collective tasks differ from the map and reduce operations. Reduce functions, however, may be adapted to serve as intermediate processing functions in the hierarchical group model.

### **5.1.2 Programming Interfaces**

As previously discussed, most programming models fit into one of three classes (message-based, restrictive, and global). However, models that target specific applications can include different elements of these three classes. For example, EnviroTrack [10] uses mechanisms similar to logical groups. EnviroTrack, however, is designed primarily for mobile object tracking applications. As such, the tasking and communication abstractions are not as general as those provided by the hierarchical group model.

Another system, SINA [90], automatically clusters sensor nodes based on power level and geographic proximity. However, clusters that rely on other attributes, such as shared sensor values, can only be defined implicitly. Römer et al. have explored frameworks for role assignment in sensor networks [87]. Although role assignment and tasking are related subjects, the hierarchical group model operates at a lower level in the software stack and provides complementary services.

Finally agent-based systems, such as Agilla [43, 42], offer advantages over existing message-based models by substituting messages containing data with messages containing computation (i.e. agents). However, these abstractions replace the underlying computational abstraction (events, threads, and functions) without abstracting common computational and communication structures prevalent in sensor network applications. As such the hierarchical group model can provide additional support to these systems to simplify tasking and programming.

### 5.1.3 Debugging and Management Interfaces

Exploratory work has been performed on debugging and management interfaces for sensor networks. These interfaces are also used to collect and view data from the sensor network. Unlike Kensho and Tables, these interfaces often lack an underlying programming model. As a consequence, these systems provide only rudimentary programming capabilities. For example, MoteView [98] is a graphical interface in which users can only manipulate existing application parameters. Other similar web based tools include Microsoft SensorWeb [76] and SensorBase [20]. Besides data management, these tools also provide online collaboration features.

Marionette [103] is a more advanced debugging system that provides interactive debugging support for TinyOS programs. Users are able to probe for data values and invoke functions on the sensor node. This system can also be used to prototype applications. Marionette transports debugging values from the sensor node to the basestation, which in turn executes the code that normally runs on the sensor node. Although useful for debugging TinyOS programs, Marionette does not fundamentally alter the TinyOS programming model. Other systems that target TinyOS include Viptos [22] and TOSDev [72]. These systems act primarily as a development environment for NesC.

Finally, recent work by the Arch Rock Corporation includes tools for incorporating sensor networks into existing network infrastructures [30]. Their software allows users to manage and debug a sensor network using existing network analysis tools, such as *ping* and *traceroute*, using an intermediate TCP/IP proxy. While their work emphasizes integrating sensor networks into existing network namespaces, their work does not include new programming methods. Instead users are expected to program with existing TinyOS-based tools.

### 5.1.4 Privacy

An important element in urban sensor network applications is the need to protect confidential data. As previously discussed, both the negative survey and the negative database provide protection mechanisms within the constraints of the hierarchical group model. In this section, additional privacy-preserving algorithms are described and analyzed with respect to the hierarchical group model.

Data privacy is usually achieved with cryptographic techniques [94, 91]. Recent work shows that it is possible to use encryption techniques on existing sensor platforms [100, 62]. However, the computational costs are still relatively large. Also many of the proposals rely on key distribution [36]. Complex key distribution protocols often do not share the same communication patterns as the rest of the sensor network application. This makes integration into existing applications difficult.

Secure multiparty computation algorithms allow nodes to compute any function of many variables without each node knowing the inputs of the other nodes [89]. For instance, secure multiparty algorithms can be used to calculate the average salary of a group of people without the individuals learning the actual salary of each person. These algorithms, however, require cryptographic methods and often require synchronized communication.

In general, cryptographic systems do not integrate well into existing sensor network programming models, largely due to the complexity of key distribution protocols. Although most applications have a regular hierarchical communication structure, many key distribution protocols require complex local communication. Consequently, developing new protocols alongside applications may be difficult. In contrast, algorithms that share the same communication structure as applications, such as the negative survey and the location anonymity algorithm, are much easier to integrate into application frameworks.

There has also been recent work on generic privacy frameworks. AnonySense [29] is designed for use with personal devices (such as phones) within urban areas. The sys-

## *Chapter 5. Related Work and Conclusion*

tem includes a tasking language that specifies the type of data to be collected from these devices, and anonymizes “reports” using a MIX network [21]. However, their work is concerned with anonymizing the source of the data and does not address the data itself. Consequently, it is unclear how their tasking language can be used within the context of other programming environments.

Recent work on participatory, urban sensing addresses the need for application specific privacy concerns [85, 95]. In this scheme, specified servers would act to sanitize sensitive data in application specific ways. For example, the resolution of location data could be reduced depending on the application and the user. These schemes, however, do not exploit the ability for sensor nodes to protect confidential data.

## **5.2 Future Work**

Future work in this area includes possible extensions to the hierarchical group model and new implementation strategies for collective tasks. Currently in Kensho and Tables collective tasks execute on the basestation, while local tasks execute on the sensor nodes. This is true regardless of the size and topology of the sensor network. Although this is a straightforward implementation of collective tasks, the hierarchical group model does not preclude alternative implementations. There are several possible implementations to explore in the future. For example, new implementations can focus on reducing collective task latency or the total number of message transmissions. This can be accomplished by executing the collective tasks inside the sensor network. There several possible strategies to accomplish this, some of which are enumerated here.

1. Local leaders: In this scheme, sensor nodes that detect an event would initially contact the basestation with routing information. This routing information would then be used to select a leader that is able to intercept all publish messages within



## *Chapter 5. Related Work and Conclusion*

the existing routing scheme. Ideally, the selected leader would also be closer to the phenomenon (in terms of hop count) than the basestation. This leader would stop forwarding the publish messages and perform the collective computation. Since the leader is closer to the event, the number of total message transmissions would be lower. However, this scheme assumes that the underlying routing structure is stable and an appropriate sensor node can be found.

2. **Pre-selected leaders:** In this scheme, a number of sensor nodes are selected during the initialization stage to serve as leaders. A sensor node, upon detecting an event, will transmit a publish message towards the basestation. If there is a pre-selected leader along the routing path, the leader will intercept the message and execute the collective task. In the case that a leader is not found along the routing path, the basestation will eventually receive the message and act as a default leader. This particular scheme avoids the problem of having to dynamically elect a leader. As long as there are enough pre-selected leaders, most events will be covered by a nearby leader. However, it is possible that two sensor nodes that are geographically nearby may contact two different leaders due to the underlying routing structure.
3. **Multiple routing:** In the other two schemes, the underlying routing structure was used to determine the leader of a logical group. In this scheme, sensor nodes belonging to a logical group would elect a new local leader. This is accomplished by first listening for appropriate leader beacons, and then starting an election in the case that a beacon is not heard. The elected leader, in turn, will create a new routing tree for that particular logical group. This scheme ensures that even if the underlying routing structure changes, most publish and push messages will reach the correct destination. However, this scheme involves a relatively complex election and group construction phase that increases the initial latency and transmission count.

These in-network leader schemes can also integrate heterogeneous sensor nodes. Leaders that are more computationally powerful or have a wider communication range may

## *Chapter 5. Related Work and Conclusion*

serve as better leaders. Besides lowering latency and reducing the number of transmissions, alternative implementations can also focus on fault-tolerance. For example, instead of having a single leader (either the basestation or sensor node), the implementation may use multiple sensor nodes to share collective tasks. By scheduling certain collective tasks to certain sensor nodes, the overall lifetime of each leader can be extended (since each leader is performing fewer operations). Similarly, if a few of the leaders fail, only a subset of the collective tasks will fail. Other multiple leader strategies include only having a single leader at any one time but scheduling these duties across multiple sensor nodes. Such a scheme would ensure that collective operations continue to function even as potential leaders fail. These multi-leader schemes, however, assume that there are either several collective tasks that need to be scheduled or that a basestation is inaccessible (since performing the computation on the basestation is most likely the safest strategy).

Finally, new implementations can also focus on data redundancy. Sensor nodes, upon collecting data, would selectively distribute this data over a subset of the sensor network. This ensures that if a collective task requests data from a malfunctioning sensor node, the data can be found in other locations. Even when the sensor nodes do not malfunction, preemptively distributing data may result in lower collective task latency. If a sensor node can predict how often some data will be requested, the data can be pushed closer to the root of the routing tree to ensure that the collective task receives the data with fewer hops.

Besides typical sensor network environments, the hierarchical group model can also be adapted for other computational settings. For example, both passive and active radio frequency identification devices (RFID) can be used in many sensing environments. However, these devices are more constrained than typical sensor platforms with very little memory and computational capability. As a consequence, even storing and executing local tasks may exceed the abilities of such hardware. In that case, the hierarchical group implementation would need to move local tasks to more capable machines such as a basestation. The RFID devices would simply transmit the data they sense. So long as applications em-

## Chapter 5. Related Work and Conclusion

ploy a standard API, such as Kensho, the application would be oblivious to where the actual computation took place.

Other environments, such as mobile personal devices (phones or ultra-portable computers) have more memory, computational capability, and communication range than typical sensor platforms. These devices may also operate over a larger area and use the internet as the default communications network. The hierarchical group model can also be adapted for these devices. Phones can be equipped with local tasks that communicate with collective tasks via the internet. These collective tasks, in turn, would execute on various internet servers. Logical groups would be used to dictate when and where these devices operate.

Once application developers construct a sensor network application using the hierarchical group model, the developer must still decide which collective task implementation to use. This decision will depend on the characteristics of the environment along with application requirements. For example, applications using computationally limited devices may choose to execute all tasks (local and collective) on the basestation. Other applications may only have a limited bandwidth connecting the basestation to the sensor network. For these applications both collective and local tasks should execute on the sensor network to minimize communication. An important topic to explore in the future is whether these application requirements and environmental constraints can be used to *automatically* select the appropriate hierarchical group implementation.

The environmental model used to describe environmental constraints should include a description of the major components (such as the sensor nodes and basestation). These components include memory, computational capability, and storage capabilities. The model should also include the availability of links between major components and describe the bandwidth, latency, and costs associated with these links. The application requirements should include quality-of-service requirements, cost and data storage requirements, and expected monetary budgets. Given these requirements, a mapping system can take advantage of the hierarchical group model to decide where to execute collective and

## Chapter 5. Related Work and Conclusion

local tasks. For example, if the application developer specifies that all the data collected by the sensor network should be permanently stored, the system should find a hardware component that is able to store all the data (perhaps by transmitting all the data back to the basestation).

Although the hierarchical group model accurately characterizes many sensor network applications, the model does not consider the statistical confidence of collected data. For example, when the user collects photometer data, the accuracy and confidence of that data is not returned. For collective tasks that benefit from such information (such as object tracking), the user must implement these functions manually. Integrating accuracy and confidence into the data communication mechanisms may benefit many applications. By combining these mechanisms with user supplied confidence requirements, the system may also be able to operate more efficiently (perhaps by limiting the logical group size). However, these mechanisms must be implemented carefully, since accuracy and confidence are often application specific.

### 5.3 Conclusion

This dissertation introduced the hierarchical group model. In this model, computation is tasked to logical groups and split into collective and local components that communicate hierarchically. Local computation is primarily used for data production and *publishes* data to the collective computation. Similarly, collective computation is primarily used for data aggregation and *pushes* results back to the local computation. These abstractions decouple computation from hardware and can characterize and aid in the construction of sensor network software with minimal overhead.

To validate the hierarchical group model, this dissertation introduced two implementations of the model. Kensho is a C-based implementation of the hierarchical group

## *Chapter 5. Related Work and Conclusion*

model that can be used for a variety of user applications. Another implementation, Tables, presents a spreadsheet-inspired view of the sensor network that takes advantage of hierarchical groups for both computation and communication. Users are able to specify both local and collective functions that execute on the sensor network via the spreadsheet interface.

This dissertation also introduced several applications that use the hierarchical group model to organize computation and communication. One application, FUSN, provides a set of methods for constructing filesystem-based interfaces for sensor networks. Another set of applications included novel privacy algorithms that use negative representations of data to anonymize data collection and preserve location anonymity.

Future sensor network applications will move beyond static, homogeneous deployments to include dynamic, heterogeneous elements. These sensor networks will also gain new users, including casual users who will expect intuitive interfaces to interact with sensor networks. The hierarchical group model and the associated implementations of the model address these challenges and bring sensor networks closer to the realm of practical scientific tools.

# References

- [1] Citysense: <http://www.sensenetworks.com/citysense.php>.
- [2] Collection tree protocol: <http://www.tinyos.net/tinyos-2.x/doc/txt/tep123.txt>.
- [3] Filesystem in userspace: <http://fuse.sourceforge.net/>.
- [4] GGA: <http://aprs.gids.nl/nmea>.
- [5] <http://campaignr.com/>.
- [6] Jessica's law: [http://en.wikipedia.org/wiki/jessica's\\_law](http://en.wikipedia.org/wiki/jessica's_law).
- [7] Mologogo: <http://www.mologogo.com>.
- [8] Princeton: zchaff. <http://ee.princeton.edu/~EIJchaff/zchaff.php>.
- [9] Weather underground: <http://www.wunderground.com>.
- [10] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. Envirotrack: Towards an environmental computing paradigm for distributed sensor networks. In *International Conference on Distributed Computing Systems (ICDCS)*, 2004.
- [11] H. Abrach, S. Bhatti, J. Carlson, H. Dai, J. Rose, A. Sheth, B. Shucker, and R. Han. Mantis: System support for multimodal networks of in-situ sensors. In *Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2003.
- [12] R. Agrawal and R. Srikant. Privacy-preserving data mining. In *ACM SIGMOD Conference on Management of Data*, 2000.

## References

- [13] K. Ara, N. Kanehira, D. O. Olguin, B. N. Waber, T. Kim, A. Mohan, P. Gloor, R. Laubacher, D. Oster, A. S. Pentland, and K. Yano. Sensible organizations: Changing our businesses and work styles through sensor data. *Journal of Information Processing (Information Processing Society of Japan)*, 16(April), 2008.
- [14] T. Berger-Wolf, S. Sheikh, B. DasGupta, M. Ashley, I. Caballero, W. Chaovalitwongse, and S. L. Putrevu. Reconstructing sibling relationships in wild populations. *Bioinformatics*, 23(13):49–56, 2007.
- [15] S. M. Brennan, A. M. Mielke, and D. C. Torney. Radioactive source detection by sensor networks. *IEEE Transactions on Nuclear Science*, 52:813–819, 2005.
- [16] S. M. Brennan, A. M. Mielke, D. C. Torney, and A. Maccabe. Radiation detection with distributed sensor networks. *IEEE Computer*, 34:57–59, 2004.
- [17] J. Burke, D. Estrin, M. Hansen, A. Parker, N. Ramanathan, S. Reddy, and M. B. Srivastava. Participatory sensing. In *Workshop on World-Sensor-Web (SenSys)*, 2006.
- [18] A. Cerpa, J. Elson, D. Estrin, L. Girod, M. Hamilton, and J. Zhao. Habitat monitoring: Application driver for wireless communications technology. In *ACM SIGCOMM Workshop on Data Communications in Latin America and the Caribbean*, 2001.
- [19] H. Chan, A. Perrig, and D. Song. Secure hierarchical in-network aggregation in sensor networks. In *International Conference on Computer and Communications Security (CCS)*, 2006.
- [20] K. Chang, N. Yau, M. Hansen, and D. Estrin. Sensorbase.org - a centralized repository to slog sensor network data. In *Euro-American Workshop on Middleware for Sensor Networks (EAWMS - DCOSS)*, 2006.
- [21] D. Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.
- [22] E. Cheong, E. A. Lee, and Y. Zhao. Viptos: a graphical development and simulation environment for tinyos-based wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.
- [23] S. Y. Cheung, S. C. Ergen, and P. Varaiya. Traffic surveillance with wireless magnetic sensors. In *12th World Congress on Intelligent Transport Systems*, 2005.
- [24] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov. Characteristics of workloads used in high performance and technical computing. In *International Conference on Supercomputing (ICS)*, 2007.

## References

- [25] D. Chu, K. Lin, A. Linares, G. Nguyen, and J. Hellerstein. Sdlib: A sensor network data and communications library for rapid and robust application development. In *Information Processing in Sensor Networks (IPSN)*, 2006.
- [26] D. Chu, L. Popa, A. Tavakoli, J. Hellerstein, P. Levis, S. Shenker, and I. Stoica. The design and implementation of a declarative sensor network system. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.
- [27] M. Colagrosso, W. Simmons, and M. Graham. Demo abstract: Simple sensor syndication. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.
- [28] UPC Consortium. UPC language specifications, v1.2, lbnl-59208. Technical report, Lawrence Berkeley National Lab, 2005.
- [29] C. Cornelius, A. Kapadia, D. Kotz, D. Peebles, M. Shin, and N. Triandopoulos. Anonymsense: An architecture for privacy-aware urban sensing. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2008.
- [30] A. R. Corporation. A sensor network architecture for the ip enterprise. In *Information Processing in Sensor Networks (IPSN)*, 2007.
- [31] J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. 2004.
- [32] P. P. Dey, S. Chandra, and S. Gangopadhaya. Speed distribution curves under mixed traffic conditions. *Journal of Transportation Engineering*, 132, 2006.
- [33] A. Dunkels, N. Finne, J. Eriksson, and T. Voigt. Run-time dynamic linking for reprogramming wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.
- [34] A. Dunkels, B. Gronvall, and T. Voigt. Contiki - a lightweight and flexible operating system for tiny networked sensors. In *IEEE International Conference on Local Computer Networks (LCN)*, 2004.
- [35] S. C. Ergen, S. Y. Cheung, P. Varaiya, R. K. Iyer, and A. Haoui. Demonstration: Wireless sensor networks for traffic monitoring. In *IPSN*, 2005.
- [36] L. Eschenauer and V. D. Gligor. A key-management scheme for distributed sensor networks. In *ACM Conference on Computer and Communications Security (CCS)*, 2002.
- [37] F. Esponda. Negative surveys. *ArXiv Mathematics e-prints*, Aug 2006.



## References

- [38] F. Esponda. Hiding a needle in a haystack using negative databases. In *Information Hiding*, 2008.
- [39] F. Esponda, E. S. Ackley, P. Helman, H. Jia, and S. Forrest. Protecting data privacy through hard-to-reverse negative databases. *International Journal of Information Security*, 6(6), 2007.
- [40] D. Estrin, R. Govindan, J. Heidemann, and S. Kumar. Next century challenges: scalable coordination in sensor networks. In *International Conference on Mobile Computing and Networking (MobiCom)*, 1999.
- [41] R. Finkel and J. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4(1):1–9, 1974.
- [42] C.-L. Fok, G.-C. Roman, and C. Lu. Mobile agent middleware for sensor networks: An application case study. In *Information Processing in Sensor Networks (IPSN)*, 2005.
- [43] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *International Conference on Distributed Computing Systems (ICDCS)*, 2005.
- [44] R. Fonseca, O. Gnawali, K. Jamieson, and P. Levis. Four-bit wireless link estimation. In *Hot Topics in Networks (HotNets)*, 2007.
- [45] M. P. I. Forum. Mpi: A message-passing interface standard. *International Journal of High Performance Computing Applications*, 8(3):165–414, 1994.
- [46] D. Gay, P. Levis, R. von Behren, M. Welsh, E. Brewer, and D. Culler. The nesc language: A holistic approach to networked embedded systems. In *Programming Language Design and Implementation (PLDI)*, 2003.
- [47] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. S. Sunderam. *PVM: Parallel Virtual Machine A Users' Guide and Tutorial for Network Parallel Computing*. MIT Press, Cambridge, Massachusetts, 1994.
- [48] L. Girod, M. Lukac, V. Trifa, and D. Estrin. The design and implementation of a self-calibrating distributed acoustic sensing platform. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.
- [49] O. Gnawali, B. Greenstein, K.-Y. Jang, A. Joki, J. Paek, M. Vieira, D. Estrin, R. Govindan, and E. Kohler. The tenet architecture for tiered sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.

## References

- [50] R. Govindan. The quest for a general-purpose sensing system. In *Keynote: Workshop on Embedded Networked Sensors (EmNets)*, 2007.
- [51] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *International Conference on Distributed Computing in Sensor Systems (DCOSS)*, 2005.
- [52] C.-C. Han, R. Kumar, R. Shea, E. Kohler, and M. Srivastava. Sos: A dynamic operating system for sensor nodes. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2005.
- [53] J. Hill and D. Culler. A wireless embedded sensor architecture for system-level optimization. Technical report, University of California - Berkeley, 2002.
- [54] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. E. Culler, and K. S. J. Pister. System Architecture Directions for Networked Sensors. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2000.
- [55] J. Horey, P. Bridges, A. Maccabe, and A. Mielke. Work-in-progress: The design of a spreadsheet interface. In *Information Processing in Sensor Networks (IPSN)*, 2005.
- [56] J. Horey, M. Groat, S. Forrest, and F. Esponda. Anonymous data collection in sensor networks. In *International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services (MobiQuitous)*, 2007.
- [57] B. Hull, V. Bychkovsky, Y. Zhang, K. Chen, M. Goraczko, A. Miu, E. Shih, H. Balakrishnan, and S. Maadden. Cartel: A distributed mobile sensor computing system. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.
- [58] L. B. Huston and P. Honeyman. Disconnected operation for afs. In *Mobile & Location-Independent Computing Symposium (MLCS)*, 1993.
- [59] H. Jia. *What Makes NP-Complete Problems Hard?* PhD thesis, University of New Mexico, 2007.
- [60] P. Juang, H. Oki, Y. Wang, M. Martonosi, L.-S. Peh, and D. Rubenstein. Energy-Efficient Computing for Wildlife Tracking: Design Tradeoffs and Early Experience with ZebraNet. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.
- [61] J. M. Kahn, R. H. Katz, and K. S. J. Pister. Next century challenges: mobile networking for smart dust. In *International Conference on Mobile Computing and Networking (MobiCom)*, 1999.

## References

- [62] C. Karlof, N. Sastry, and D. Wagner. Tinysec: A link layer security architecture for wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2004.
- [63] S. Kim, R. Fonseca, P. Dutta, A. Tavakoli, D. Culler, P. Levis, S. Shenker, and I. Stoica. Flush: a reliable bulk transport protocol for multihop wireless networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2007.
- [64] S. Kim, S. Pakzad, D. Culler, J. Demmel, G. Fenves, S. Glaser, and M. Turon. Health monitoring of civil infrastructures using wireless sensor networks. In *Information Processing in Sensor Networks (IPSN)*, 2007.
- [65] J. J. Kistler and M. Satyanarayanan. Disconnected operation in the coda file system. *ACM Transactions on Computer Systems*, 10(1):3–25, 1992.
- [66] U. Lee, E. Magistretti, B. Zhou, M. Gerla, P. Bellavista, and A. Corradi. Mobeyes: Smart mobs for urban monitoring with a vehicular sensor network. In *IEEE Wireless Communications*, 2006.
- [67] P. Levis and D. Culler. Maté: A Tiny Virtual Machine for Sensor Networks. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, 2002.
- [68] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. Tinydb: an acquisitional query processing system for sensor networks. *ACM Transaction Database Systems*, pages 122–173, 2005.
- [69] G. Mainland and M. Welsh. Programming sensor networks using abstract regions. In *Symposium on Networked Systems Design and Implementation (NSDI)*, 2004.
- [70] A. Mainwaring, J. Polastre, R. Szewczyk, D. Culler, and J. Anderson. Wireless Sensor Networks for Habitat Monitoring. In *Workshop on Wireless Sensor Networks and Applications (WSNA)*, 2002.
- [71] D. Malan, T. Fulford-Jones, M. Welsh, and S. Moulton. Codeblue: An ad hoc sensor network infrastructure for emergency medical care. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.
- [72] W. P. McCartney and N. Sridhar. Tosdev: a rapid development environment for tinyos. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2006.
- [73] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Design Automation Conference (DAC)*, 2001.

## References

- [74] M. Mun, D. Estrin, J. Burke, and M. Hansen. Parsimonious mobility classification using gsm and wifi traces. In *Workshop on Embedded Networked Sensors (HotEm-Nets)*, 2008.
- [75] R. Murty, G. Mainland, I. Rose, A. R. Chowdhury, A. Gosain, J. Bers, and M. Welsh. Citysense: A vision for an urban-scale wireless networking testbed. In *IEEE International Conference on Technologies for Homeland Security*, 2008.
- [76] S. Nath, J. Liu, and F. Zhao. Sensormap for wide-area sensor webs. *IEEE Computer Magazine*, 40(7):90–93, 2007.
- [77] R. Newton, G. Morrisett, and M. Welsh. The regiment macroprogramming system. In *Information Processing in Sensor Networks (IPSN)*, 2007.
- [78] Y. Ni, U. Kremer, and L. Iftode. Spatial views: Space-aware programming for networks of embedded systems. In *Workshop on Languages and Compilers for Parallel Computing*, 2003.
- [79] A. Perrig, R. Szewczyk, V. Wen, D. Culler, and J. D. Tygar. Spins: Security protocols for sensor networks. In *International Conference on Mobile Computing and Networking (MobiCom)*, 2001.
- [80] R. Pike, D. Presotto, S. Dorward, B. Flandrena, K. Thompson, H. Trickey, and P. Winterbottom. Plan 9 from Bell Labs. *Computing Systems*, 8(3):221–254, Summer 1995.
- [81] B. Pisupati and G. Brown. Poster: File system framework for organizing sensor networks. In *Symposium on Applied Computing (SAC)*, 2006.
- [82] N. B. Priyantha, A. Chakraborty, and H. Balakrishnan. The cricket location-support system. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2000.
- [83] B. Przydatek, D. Song, and A. Perrig. Sia: Secure information aggregation in sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.
- [84] M. Rahimi, R. Baer, O. I. Iroezi, J. C. Garcia, J. Warrior, D. Estrin, and M. Srivastava. Cyclops: In situ image sensing and interpretation in wireless sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.
- [85] S. Reddy, G. Chen, B. Fulkerson, S. J. Kim, U. Park, N. Yau, J. Cho, M. Hansen, and J. Heidemann. Sensor-internet share and search: Enabling collaboration of citizen scientists. In *Workshop for Data Sharing and Interoperability - IPSN*, 2007.

## References

- [86] S. Reddy, A. Parker, J. Hymanv, J. Burke, M. Hansen, and D. Estrin. Image browsing, processing, and clustering for participatory sensing: Lessons from a dietsense prototype. In *Workshop on Embedded Networked Sensor (EmNets)*, 2007.
- [87] K. Römer, C. Frank, P. J. Marrón, and C. Becker. Generic role assignment for wireless sensor networks. In *ACM SIGOPS European Workshop*, 2004.
- [88] M. Satyanarayanan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, April 1990.
- [89] B. Schneier. *Applied Cryptography Second Edition*. John Wiley and Sons, Inc., 1996.
- [90] P.-C. Shen, C. Srisathapornphat, and C. Jaikaeo. Sensor Information Networking Architecture and Applications. *IEEE Personel Communication Magazine*, August 2001.
- [91] E. Shi, J. Bethencourt, H. Chan, D. Song, and A. Perrig. Multi-dimensional range query over encrypted data. In *IEEE Security and Privacy Symposium (SP)*, 2007.
- [92] E. Shi and A. Perrig. Designing secure sensor networks. *IEEE Wireless Communications*, 2004.
- [93] S. Y. C. Sinem Coleri and P. Varaiya. Sensor networks for monitoring traffic. In *Allerton Conference on Communication, Control and Computing*, 2004.
- [94] D. X. Song, D. Wagner, and A. Perrig. Practical techniques for searches on encrypted data. In *ACM on Computer and Communications Security (CCS)*, 2000.
- [95] M. Srivastava, M. Hansen, J. Burke, A. Parker, S. Reddy, T. Schmid, K. Chang, G. Saurabh, M. Allman, V. Paxson, and D. Estrin. Network system challenges in selective sharing and verification for personal, social, and urban-scale sensing applications. In *Workshop on Hot Topics in Networks (HotNets)*, 2006.
- [96] S. Tilak, B. Pisupati, K. Chiu, G. Brown, and N. Abu-Ghazaleh. A file system abstraction for sense and respond systems. In *Workshop on End-to-End, Sense-and-respond Systems, Applications, and Services (EESR)*, 2005.
- [97] G. Tolle, J. Polastre, R. Szewczyk, D. Culler, N. Turner, K. Tu, S. Burgess, T. Dawson, P. Buonadonna, D. Gay, and W. Hong. A macroscope in the redwoods. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2005.
- [98] M. Turon. Mote-view: A sensor network monitoring and management tool. In *Workshop on Embedded Networked Sensors (EmNets)*, 2005.

## References

- [99] T. von Eicken, D. E. Culler, S. C. Goldstein, and K. E. Schauer. Active messages: A mechanism for integrated communication and computation. *ACM SIGARCH Computer Architecture News*, 20(2):256–266, 1992.
- [100] H. Wang, B. Sheng, and Q. Li. Elliptic curve cryptography-based access control in sensor networks. *International Journal of Security and Networks*.
- [101] G. Werner-Allen, J. Johnson, M. Ruiz, J. Lees, and M. Welsh. Monitoring volcanic eruptions with a wireless sensor network. In *European Workshop on Wireless Sensor Networks*, 2005.
- [102] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: a neighborhood abstraction for sensor networks. In *International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2004.
- [103] K. Whitehouse, G. Tolle, J. taneja, C. Sharp, S. Kim, J. Jeong, J. Hui, P. Dutta, and D. Culler. Marionette: Using rpc for interactive development and debugging of wireless embedded networks. In *Information Processing in Sensor Networks (IPSN)*, 2006.
- [104] A. Woo, S. Seth, T. Olson, J. Liu, and F. Zhao. A spreadsheet approach to programming and managing sensor networks. In *Information Processing in Sensor Networks (IPSN)*, 2006.
- [105] A. Woo, T. Tong, and D. Culler. Taming the underlying challenges of reliable multihop routing in sensor networks. In *ACM Conference on Embedded Networked Sensor Systems (SenSys)*, 2003.
- [106] Y. Yao and J. Gehrke. The Cougar Approach to In-Network Query Processing in Sensor Networks. In *ACM SIGMOD Conference*, 2002.
- [107] S. Zhang, J. Ford, and F. Makedon. Deriving private information from randomly perturbed ratings. In *Siam Conference on Data Mining*, 2006.