SCALABLE QUERY PROCESSING IN SERVICE-ORIENTED SENSOR NETWORKS

By

RAJA BOSE

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

2009

To my grandfather, Samarendra Kumar Mitra, who designed and built India’s first computer

ACKNOWLEDGMENTS

I thank my advisor Dr. Abdelsalam (Sumi) Helal for his constant help, guidance and support. I thank all the members of the Mobile and Pervasive Computing Laboratory, both past and present who were involved in design and development of the Atlas Platform and associated middleware. I especially thank Hen-I Yang for all his help during development of the Virtual Sensors Framework and its experimental evaluation. I thank Steven Vanderploeg, James Russo, and Steven Pickles for their role in designing, building, and testing the Atlas hardware. I express my utmost gratitude to Chao Chen for his constructive suggestions and debugging help.

TABLE OF CONTENTS

page

LIST OF TABLES

LIST OF FIGURES

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

SCALABLE QUERY PROCESSING IN SERVICE-ORIENTED SENSOR NETWORKS

By

Raja Bose

May 2009

Chair: Abdelsalam (Sumi) Helal
Major: Computer Engineering

The widespread availability of sensor devices and the rapid increase in their deployment, everywhere from industrial plants to private homes has put sensor network research in the spotlight for the past several years. Moreover, the requirement for rich highly configurable sensor network applications has led to the emergence of Service Oriented Sensor Networks (SOSNs), which imports the concept of Service Oriented Architecture (SOA) into the sensor network domain. It represents each of its sensors as a service object in a service framework that allows their dynamic discovery and composition into applications. Representing sensors as composable services and utilizing their associated knowledge can lead to significant enhancements in information processing capabilities of a sensor network, allowing it to operate on sophisticated data types and events beyond the primitive data types typically originating from individual hardware sensors. This dissertation describes the research and development of *Sensable*, a scalable query processing middleware which extends the capabilities of service-oriented sensor networks as follows: (1) Provides adaptive sensor-aware query processing to minimize overall power consumption in sensor networks by utilizing knowledge associated with sensor services to identify and minimize sensing operations which cause significant power drain; (2) Enhances Smart Space capabilities to sense virtual types of data which cannot be directly

sourced from physical sensors due to their inherent sophistication or higher Quality of Service (QoS) requirements and; (3) Enables Smart Spaces to monitor and track phenomena event clouds whose shape, size and motion cannot be modeled precisely.

Sensable attempts to bring query processing using service-oriented sensor networks into the realm of immediate utility by providing query scalability based on actual network infrastructure using current and emerging technologies and advanced querying capabilities encompassing abstract data types fused from multiple sensors, the concept of data quality in face of failure and detection and tracking of events susceptible to uncertainty in face of noise.

CHAPTER 1
INTRODUCTION

The requirement for rich, highly-configurable sensor network applications led to the emergence of Service Oriented Sensor Networks (SOSN). A SOSN imports the concept of Service Oriented Architecture (SOA) into the sensor network domain. It represents each of its sensors as a service object in a service framework that allows their dynamic discovery and composition into applications [9]. This provides a loosely-coupled model where there is a strict separation between application logic and device-specific operational logic. In such a system, sensors are not integrated using hard-coded routines inside applications but instead are discovered, integrated and accessed on-demand. Moreover, such loose-coupling also enables multiple applications to simultaneously share the same set of devices deployed in the space without interfering with each others' operations.

The representation of a sensor as a service immediately opens up a number of possibilities for enhancing the information processing capabilities of the sensor network. Each sensor is now associated with some knowledge about the device which can be exploited to provide more power-efficient query processing in the network. Representing individual sensors as services in an SOA can also provide the ability to dynamically compose multiple data sources either to derive more sophisticated data types which cannot be obtained directly from physical sensors or, to provide higher standards of Quality of Service (QoS) and availability guarantees. Furthermore, instead of restricting query processing to well-defined data and events, the sensor network can be made capable of monitoring and tracking phenomena events whose behavior cannot be modeled (shape, size and motion) accurately over time.

We describe *Sensable*, a scalable query processing middleware which utilizes the SOA characteristics of a SOSN to provide information processing capabilities beyond those available

in traditional query processors which only deal with primitive data directly obtained from low-level physical sensors. The middleware is designed and built on top of the Atlas Platform [18], which is a plug-and-play service-oriented sensor platform developed at the Mobile and Pervasive Computing Laboratory at the University of Florida. The middleware is composed of three distinct components (Figure 1-1):

1. Adaptive sensor-aware in-network query processing for minimizing power consumption in sensor networks utilizing low-power communications such as ZigBee.

2. Virtual Sensors framework for processing queries involving sophisticated derived data types and queries requiring QoS guarantees.

3. Phenomena Detection and Tracking for monitoring and tracking events which cannot be defined in precise terms or have their behavior (shape, size and motion) modeled accurately over time.

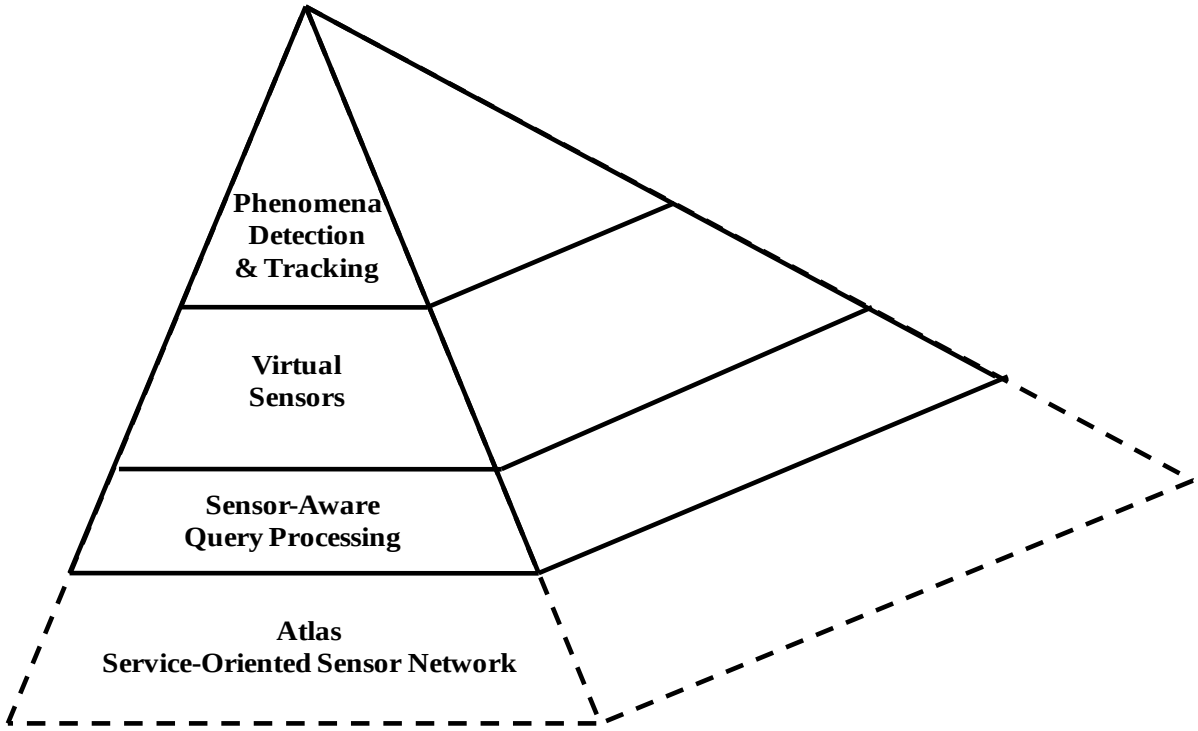

Figure 1-1. *Sensable* middleware components

The rest of this dissertation is organized as follows. In Chapter 2, we present related work. In Chapter 3, we introduce basic concepts of service oriented sensor networks and give an overview of the Atlas Platform and its hardware and software components, which provide the

basic implementation layers for *Sensable*. In Chapter 4, we describe the overall system architecture of the *Sensable* query processing middleware.

In Chapter 5, we describe sensor-aware adaptive query processing, which utilizes knowledge provided by a sensor's service representation to execute distributed, in-network queries involving multiple sensors in a power-efficient manner.

To date, sensor network research has assumed that the cost of transmitting a sensor reading over the network is much higher than the cost of sampling a sensor. However, this assumption is no longer always valid, due to availability of new generation sensor platform hardware which utilizes industry standard mesh-networking protocols such as ZigBee on top of relatively high-speed, yet low-power wireless radios. In fact, we have experimentally verified that the energy consumed for acquiring a sample from a sensor can be significantly higher than the energy consumed for transmitting its reading over the network. Hence, new querying strategies need to be formulated, which optimize the order of sampling sensors across the network in such a manner that sensors with expensive acquisition costs are not sampled unless absolutely required. We propose distributed pull-push querying mechanisms which optimize the query plan by adapting to variable costs of acquiring readings from different sensors across the network. The goal of these mechanisms is to minimize the energy consumption of nodes executing a query while ensuring that the latency of query response does not exceed user-specified bounds. We also analyze the performance of various plan options on different hardware configurations, based on their energy consumption and latency, through experiments using real-world data.

In Chapter 6, we describe the Virtual Sensors framework. Contemporary sensor network query processing systems are typically equipped to only process queries involving primitive data types which can be directly sourced from hardware sensors. The Virtual Sensors framework on

the other hand enables the query processor to handle queries involving more abstract and derived types of data fused from multiple sensors. The framework leverages SOA concepts and plug-and-play features of the Atlas Platform to allow the sensor network to automatically detect new derived sensing capabilities, whenever new physical sensors are introduced into the space. It provides capabilities for on-demand creation of virtual sensors and their lifecycle management and also allows applications to monitor sensor data quality and provides fault-tolerance mechanisms which utilize approximation algorithms to maximize the availability of sensing resources. We also address the issues of scalability and latency and propose distributed, in-network algorithms for the creation and execution of virtual sensors. We show through experiments that the proposed mechanisms provide excellent fault tolerance and data quality monitoring capabilities and result in significantly lower latency and energy consumption as compared to the centralized stream-based approaches.

In Chapter 7, we describe phenomena detection and tracking which enables the query processor to detect and track events which cannot be precisely described or modeled. The phenomena detection and tracking component extends the information processing capability of the sensor network to execute queries which involve real-time detection and tracking of groups of clustered events called phenomena clouds. Phenomena clouds are characterized by non-deterministic, dynamic variations over time, of their shape, size and direction of motion along multiple axes. This makes it difficult to apply contemporary techniques proposed for tracking moving objects using wireless sensor networks (WSNs). In the past, the utility of phenomena detection and tracking has been limited to applications such as tracking oil spills and gas clouds. However, through our collective experience over the years in a completely different deployment domain (Smart Spaces), we have discovered great utility and value in applying this concept to

accurately and efficiently observe other types of phenomena. In this dissertation, we describe distributed sensor network algorithms for in-situ detection and tracking of phenomena clouds, which utilize localized, in-network processing to simultaneously detect and track multiple phenomena clouds in a sensor space. Our algorithms not only ensure low processing and networking overhead at the centralized query processor but also minimize the number of sensors which are actively involved in the detection and tracking processes at any given time. We validate our approach using both real-life smart home applications as well as simulation experiments, which analyze the effectiveness and efficiency of our detection and tracking algorithms. We further show that our distributed algorithms also result in significant savings in resource usage and energy consumption.

CHAPTER 2
RELATED WORK

In this section, we look at past research work done in the area of sensor network query processing systems. We organize related work into three categories corresponding to each of the three main features of *Sensable*: (1), Energy-efficient query processing; (2) Virtual Sensors and; (3) Phenomena Detection and Tracking.

Energy-Efficient Query Processing

There is a large body of available research proposing techniques for querying sensor data. These approaches can be broadly divided into stream-based and acquisition-based approaches. The stream-based approaches view data originating from sensors as a series of data streams. They assume *a priori* existence of sensor data and do not factor in sensor acquisition costs. The acquisition-based techniques on the other hand closely look at ways to sample sensors and transmitting their information so that the total energy consumption is minimized. Our sensor-aware approach is more closely related to acquisition-based techniques rather than streaming methods hence, the related work covered in this sub-section reflects that.

The related work is classified into two broad categories: acquisition-based techniques for minimizing the cost of sampling sensors, and push-pull mechanisms for query execution in sensor networks. In TinyDB, Madden et al. [23] proposed algorithms for minimizing energy consumption by optimizing the order of sampling sensors and the predicates being applied on them using a series-parallel graph. However, their approach was only proposed for ordering sensors connected to the same node rather than all sensors participating in a query. This is probably due to the fact that when TinyDB was designed, network costs were much more significant than sensor sampling costs. A model-based approach (BBQ) was proposed by Deshpande et al. [8] utilizing a time-varying multivariate Gaussian model for processing sensor

queries. BBQ pulls readings whenever the model needs to be updated and also utilizes correlation among various phenomena to select sensors which are less expensive to sample. Both the above mentioned strategies only look at the tradeoffs based on the cost of sampling one sensor versus another and do not consider the tradeoff between network costs and sensor sampling costs. Hartl et al. [11] described an inference-based technique where only a subset of nodes are activated and the readings from their sensors are used to approximate readings from other sensors using Bayesian inference. Their mechanism was proposed specifically for obtaining the global average of reading values and is subject to prediction errors due to incomplete data modeling or significant fluctuation of readings. However, the combination of our approach and model-based techniques such as those listed above can potentially provide a more optimal solution in the future.

The push-pull approach for query execution has been applied by a number of query processing strategies for sensor networks. However, as we shall see, these mechanisms mainly seek to minimize the network cost, based on the assumption that it always significantly outweighs the sensor sampling cost. Trigoni et al. [32] proposed a hybrid push-pull mechanism where sensors have their readings pushed to intermediate nodes (called view nodes), from where they are pulled by the query processor based on query requirements. However, their approach only seeks to reduce the network cost and latency of answering queries and does not factor in the cost of sampling sensors (as evidenced by the fact that it uses the push approach to sample sensors). Shenker et al. [28] proposed a structured hybrid push-pull mechanism where data is pushed from sensors onto intermediate nodes and stored using geographic hash tables. The sink nodes apply the same hash function to determine where specific data is stored and use the pull approach to retrieve data. In contrast, Liu et al. [21] used an unstructured push-pull approach

(called Comb-Needle) where queries are disseminated along the horizontal lines of a sensor grid (visualized as a comb with horizontal teeth) and data is independently pushed along the vertical lines of a sensor grid (visualized as vertical needles). However, all these hybrid push-pull mechanisms mentioned above require nodes to proactively push sensor readings to intermediate nodes in the network and hence, are fundamentally different from the push-pull approach proposed in this dissertation.

In the context of actual sensor sampling, all of the above methods can be classified as mechanisms which use the push strategy which does not place a premium on sensor sampling cost. This is essentially due to the fact that they assume that network cost always outweighs sensing cost. In contrast, our sensor-aware query processing makes no such assumption and tries to avoid sampling sensors which do not contribute to the query result at a given epoch thereby, addressing the issue where sensing cost can outweigh network cost by a large margin and become the major contributor to total energy consumption.

Virtual Sensors

Sensor fusion and virtual sensors have always been actively researched topics in the sensor network community. However, the majority of publications in this area have tended to focus on developing algorithms for sensor fusion and application specific virtual sensors [10, 14, 26, 31]. However, there are some groups of researchers who have looked at virtual sensors from a systems perspective and developed mechanisms for the creation and deployment of virtual sensors. Kabadayi et al. [16] described a middleware for virtual sensors where heterogeneous sensor readings are aggregated to generate higher abstractions of data. However, they only looked at aggregation aspects of virtual sensors without addressing reliability and availability issues. Furthermore, their proposed middleware seems to be too rudimentary and can easily be superseded by native mechanisms available in service oriented architecture (SOA) frameworks

such as OSGi [19], which form the basis of SOSNs. Lewis [20] defined virtual sensor as a combination of a physical transducer along with data signal processing (DSP) elements for reliable data estimation. Both Kabadayi and Lewis have considered different aspects of virtual sensors but none of them have proposed practical distributed mechanisms for execution, data quality monitoring and fault tolerance. The Global Sensor Network (GSN) [1] is a project at EPFL, Switzerland, which aims at providing a unified framework to users for accessing real and virtual sensors. Their work comes closest to our concept of a virtual sensor framework which enables on-demand creation and execution of software sensors. However, GSN follows a stream-based centralized approach for implementing virtual sensors where physical sensors are required to stream up all their readings to the central host where virtual sensors are executed. This not only leads to increased overhead at the central host but also leads to high latency and increased power consumption. In contrast, even though we also have a centralized service layer in our architecture, we take a distributed acquisition-based approach for the internal operations of a virtual sensor and propose mechanisms which reduce latency and improve fault tolerance of virtual sensors and lower power consumption of the sensor nodes, without sacrificing the advantages of providing a user with a unified framework for accessing virtual sensors.

Unlike the systems described above, the Virtual Sensors framework described in this dissertation provides full-featured capabilities to query multiple types of virtual data with quality monitoring and fault tolerance mechanisms. Furthermore, it also enables the on-demand creation and distributed in-network execution of virtual sensors to minimize energy consumption and latency of response.

Phenomena Detection and Tracking

Nile-PDT [2, 3] is a Phenomena Detection and Tracking (PDT) framework running on top of the centralized Nile data stream management system, developed by the Indiana Center for

Database Systems (ICDS) at Purdue University. Nile-PDT was designed for detecting and tracking phenomenon clouds such as gas clouds, oil spills and chemical waste spillage. Nile-PDT uses two custom database operators, namely, SN-Scan and SN-Join to perform phenomenon detection and tracking. The SN-Scan operator scans all the sensors in the network and chooses candidate sensors which have a high probability of detecting the phenomenon. The SN-Join operator then evaluates each of these candidate sensors and checks if they join with other candidates a certain number of times and hence, are detecting a phenomenon event. Nile-PDT uses feedback control to continuously tune the SN-Scan and SN-Join parameters to maximize efficiency of the detection process. The main drawback of the Nile-PDT approach is that it takes a streaming database view of the process. It does not consider any mechanisms for controlling the flow of data at the source sensors themselves or address power consumption and network bandwidth issues inside the sensor network. Furthermore, it requires all sensors to pump readings to the SN-Scan operator to allow it to choose phenomenon candidates, which can lead to potentially massive scalability issues.

Omotayo et al. [27] have described a data harvesting framework for tracing phenomena. They proposed algorithms for maintaining a data farm on the nodes by maximum utilization of their on-board non-volatile storage, to enable backtracking to determine the cause of a phenomenon. McErlean et al. [25] proposed distributed event detection and tracking algorithms for moving objects using WSNs. Their approach involves distributed collaboration between sensors to detect an event. Each sensor which detects an event notifies neighboring sensors about its occurrence and the central base station is only notified if a certain number of sensors agree that the object is moving, that is, the event is propagated a certain number of times. However, this system assumes the prior availability of optimal ad-hoc routing mechanisms and is primarily

designed for detecting individual discrete objects with well-defined shape and size, as opposed to phenomenon clouds whose shape and size typically cannot be defined in exact terms. Chintalapudi and Govindan [7] described algorithms for detecting sensors lying closest to the edges of a phenomenon cloud. They utilized image processing and classifier techniques to determine if a sensor lies at the edge of the phenomenon or not. However, their approach not only made simplifying assumptions regarding the shape of the edges (such as whether it is a line or an ellipse), but also led to detection of a high number of false positives, since the extreme fringes of a phenomenon are more susceptible to sensor errors and rapid fluctuations.

In contrast to the above approaches, the phenomena detection and tracking algorithms described in this dissertation do not make assumptions regarding the shape, size or motion of phenomena clouds and execute in a localized, in-network fashion which make them more scalable in terms of processing, networking and energy resources as compared to contemporary stream-based techniques.

CHAPTER 3
OVERVIEW OF THE ATLAS PLATFORM

In this section we provide a brief overview of the Atlas Platform, which is a service oriented plug-and-play sensor and actuator platform developed at the University of Florida Mobile and Pervasive Computing Laboratory. Atlas provides the basic building blocks for building a SOSN, and in its most basic incarnation, consists of a set of hardware nodes and a collection of management components running inside an OSGi framework which forms the core service layer.

Hardware

The Atlas platform hardware consists of a set of nodes which provide a physical interface to sensors and actuators. Each Atlas node is a modular hardware device composed of stackable, swappable layers, with each layer providing specific functionality. A basic Atlas node consists of 3 layers: Processing Layer, Communication Layer, and Device Connection Layer (Figure 3-1).

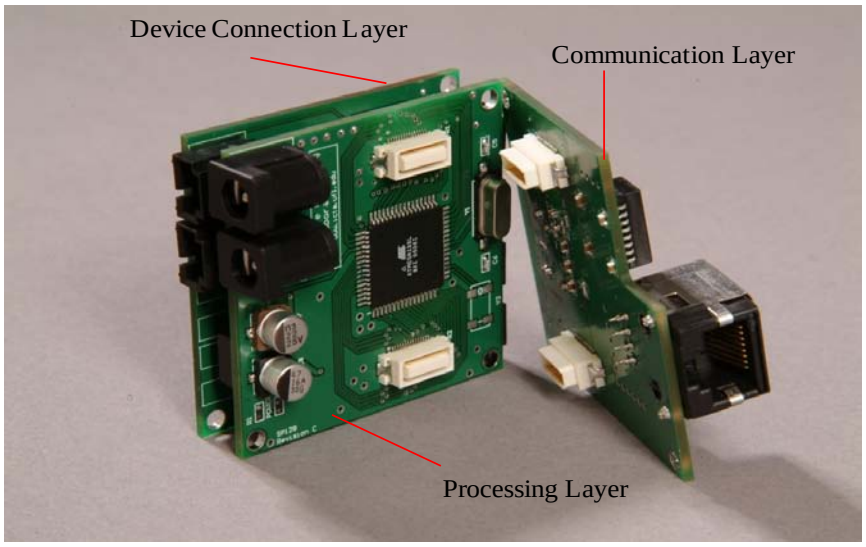

Figure 3-1. Layers of an Atlas node

The Processing Layer is based around the Atmel ATmega128L microcontroller. The ATmega128L is an 8MHz chip that includes 128KB FLASH memory and an 8-channel 10-bit

Analog-to-Digital Converter (ADC), which can operate at voltages between 2.7 and 5.5V. The Communication Layer handles the transfer of data and control-messages over the network. Currently Atlas has 4 communication options – 802.11b WiFi, wired 10BaseT Ethernet, ZigBee and USB. The Device Connection Layer is used to connect various sensors and actuators to the node. Atlas has numerous available options for connecting multiple analog and digital sensors and actuators to a single node.

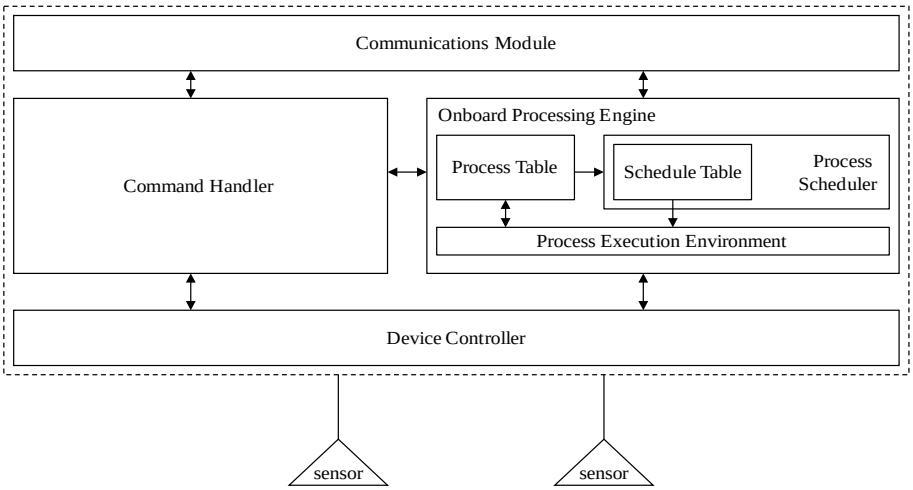

Figure 3-2. Block diagram of node firmware

Firmware

The firmware is responsible for performing four fundamental tasks, namely, controlling sensor and actuator operations, performing network communications, handling commands and control instructions and executing processes and filters pushed on the node by the service layer. Each of these tasks is handled by separate components (Figure 3-2), which abstract away the low-level details of their operation from other firmware components which may be interested in using their functionality.

Management and Service Layer

The Atlas management components run inside an OSGi service framework hosted on a central host. The host computer can be anything from a Single Board Computer (SBC) running

Linux to a full fledged standard desktop PC. OSGi is a Java-based service oriented architecture (SOA) based framework that provides a runtime environment for dynamic, transient service modules known as bundles. It provides functionalities such as life cycle management as well as service registration and discovery that are crucial for scalable composition and maintenance of applications. Each sensor or actuator is represented by Atlas as an individual OSGi bundle in the service framework. Applications are able to dynamically discover and access sensors via their respective services using standard OSGi mechanisms. The core management components of Atlas running inside the OSGi framework consist of the Network Manager, Configuration Manager and the Bundle Repository. The Network Manager handles the joining and departure of nodes in the network. The Configuration Manager manages the configuration settings of each node and enables remote configuration. The Bundle Repository stores and manages all the supported sensor and actuator service representations. Features of these three common services are also accessible to the user through an intuitive web interface.

Enabling Features for a Query Processing System

The Atlas Platform provides the following enabling features for building a query processing system for SOSNs:

1. Each sensor is represented as a service thereby abstracting away the low level operational details of the hardware from the query processor.

2. Atlas enables the query processor to access heterogeneous sensors by using generic high-level methods without having to adapt to changing sensor or node hardware specifications.

3. The plug-and-play capability of Atlas allows the query processor to be notified in case a new sensor enters the network or an old sensor suddenly goes offline. This enables the query processor to immediately take appropriate actions for queries involving the offline sensor.

4. Atlas allows different segments of the sensor network to run using diverse networking protocols, yet it provides a single unified network view to the query processor by abstracting away the routing and network management details.

5. Atlas provides distributed computing capabilities where, each Atlas node has an on-board processing engine which can execute processes pushed by the service and application layers. The query processor can push in range filters using Java method calls which are automatically translated into node instructions and automatically routed to the appropriate node.

6. Queries and applications do not have to be written as firmware code and pre-loaded on the nodes and can be pushed dynamically through the service layer. Hence, deployment of new applications or modification of existing ones do not require recompilation and updating of node firmware.

CHAPTER 4
THE *SENSABLE* QUERY PROCESSING MIDDLEWARE

In this section, we give an overview of the system architecture of the *Sensable* query processing middleware. The *Sensable* system is built on top of the Atlas Platform and consists of three major components distributed among various layers (Figure 4-1).

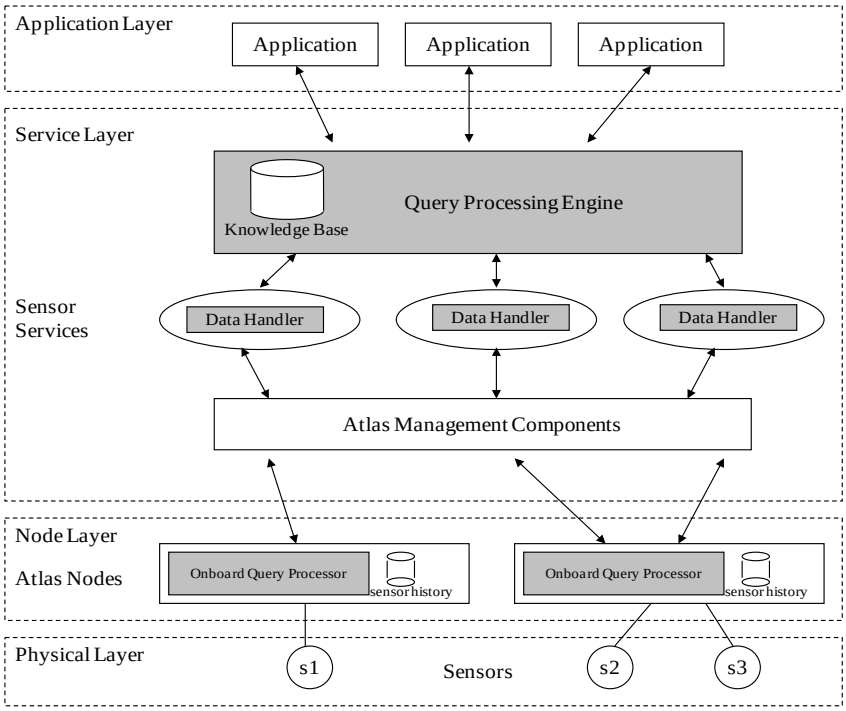

Figure 4-1. Basic architecture of *Sensable*

Query Processing Engine

The query processing engine runs as a service inside the OSGi framework. It resides in the service layer and is responsible for the overall management of all query operations in the system. It accepts queries from applications, generates optimized query plans and schedules their execution. It interacts with the sensors through their service objects and uses standard OSGi mechanisms to determine when a new sensor comes online or an existing sensor goes offline. The query plan determines whether a query needs to be executed entirely on the nodes or the processing engine or if some sub-queries have to be pushed on to the nodes while the rest are

executed centrally. The query processing engine also acts as the central clock source and can perform time synchronization between the nodes and the service layer to ensure that the nodes are in step with each other and the service layer. The knowledge base inside the query processor stores information such as phenomenon definitions, static optimization parameters and virtual sensor model definitions.

Sensor Service Data Handler

The second *Sensable* component also resides in the service layer, inside each sensor bundle. It consists of a data handler which listens for sensor readings based on epochs of queries running for that sensor. The *Sensable* query processor does not store the history of each sensor in a central location. Instead the sensor history is stored in a distributed manner at the Atlas node connected to that sensor.

Onboard Query Processor

The Onboard Processor is responsible for scheduling and executing processes which have been pushed on the node. It exists as part of the node's firmware and is responsible for executing queries pushed on to the node. It works in tandem with other node level components and performs tasks such as sampling sensors, applying filters on their data and transmitting readings back to the service layer. It also performs additional tasks such as ensuring that the node's operations are time-synchronized with the service layer and maintaining the node's portion of the distributed history of readings originating from sensors connected to it. The query processing engine can specify processes for individual sensors for tasks such as filtering out sensor data using range filters and triggering an action, in case a filter condition is satisfied. For pushing a process on a particular sensor, the *pushProcess()* method of that sensor's OSGi service, is called. The sensor service then encodes the process specifications into a process packet and forwards it to the node controlling that sensor. The node's command handler receives the message and

detects that it contains a process packet. It pushes the packet into the Process Table and notifies the Process Scheduler which decodes it and based on the process's sampling and transmission rates, reserves time slots for it in the schedule table. The Process Execution Environment looks up the Schedule Table and executes the various processes whose references are stored in it, based on the time slots allotted to them.

CHAPTER 5
SENSOR-AWARE ADAPTIVE QUERY PROCESSING

The main focus of research on query processing in sensor networks till date has been the minimization of energy consumption of the sensor network. Current research on acquisition-based query processing on sensor networks assumed that the energy cost of transmitting a sensor's reading is always significantly higher than the energy cost of sampling that sensor. However, this assumption seems to no longer be universally valid, due to rapid advances in radio hardware technology and availability of relatively high speed yet low power networking protocols such as ZigBee/802.15.4. From our experience, we found that the energy consumption of a node can be quite high even when it samples a sensor without transmitting the reading over the network and the contribution of network cost to the total energy consumption of a node can be significantly less than the sensor sampling cost. To the best of our knowledge, there is no published work which has explicitly looked at this issue and has suggested query processing strategies to deal with it.

Motivation

First, we explain why the cost of sampling a sensor can become so significant that its contribution to the total energy consumption of a node becomes extremely high as compared to network cost. Figure 5-1 shows the comparative percentage contributions of networking and sensing tasks to total energy consumption for the Crossbow MICA2 mote and the Atmel Zlink RCB sensor platform. The MICA2 was released in 2002 and has been among the first and most widely used sensor platforms in sensor network research. The Atmel Zlink RCB on the other hand, was released relatively recently in 2007 and is among a new generation of sensor platforms being developed by companies such as Atmel and Texas Instruments. The MICA2 hardware is built around an Atmel Atmega128 microcontroller and Chipcon CC1000 radio. It typically runs

the proprietary ad-hoc routing protocol used by TinyOS and has a transmission baud rate of 38.4Kbps. The Atmel Zlink RCB is based around the Atmega1281 microcontroller and AT86RF230 802.15.4 radio. It runs a full featured industry standard ZigBee stack and has a comparatively high transmission baud rate of 250Kbps. The energy consumption values for the MICA2 were derived from experiments performed by Madden et al. [23]. The energy consumption values for the Zlink RCB were calculated based on the sensor configuration used by Madden et al. [23] and hardware specifications provided by Atmel.

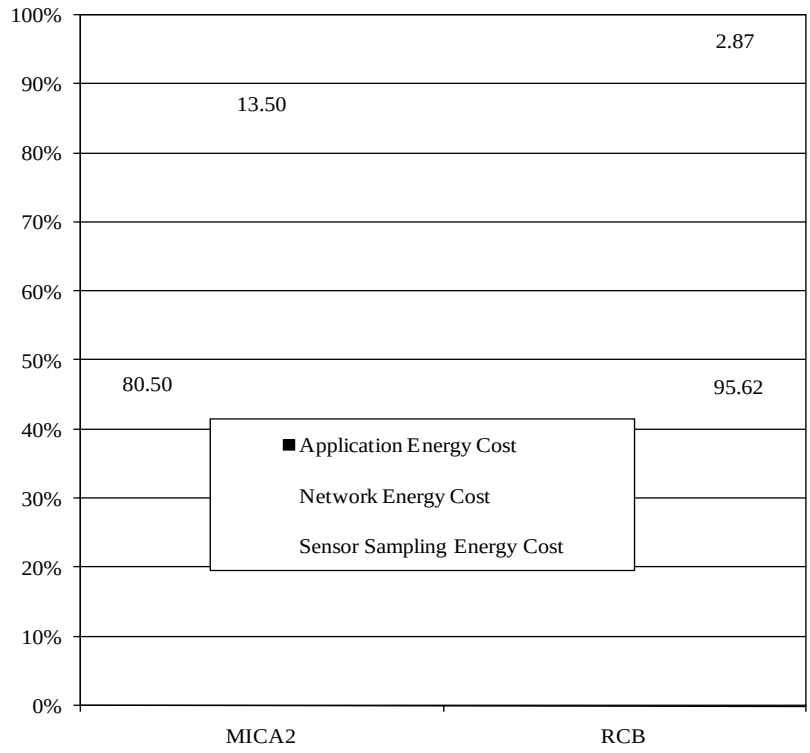

Figure 5-1. Network cost versus sensor sampling cost in MICA2 and RCB

We can observe that the contribution of sensing to the total energy consumption for the Zlink RCB is significantly higher (by about 20%) as compared to the MICA2 mote. This can be explained as follows. The newer 802.15.4 radios used by ZigBee have lower current consumption and significantly higher baud rates (250Kbps) as compared to older low-power radios such as the CC1000 (38.4Kbps) . Hence, 802.15.4 radios take much less time to transmit a

packet and therefore consume much less energy. On the other hand, the technology used for acquiring readings from a sensor has more or less remained the same. Apart from a few power hungry sensors such as magnetometers or organic byproduct sensors, most sensors have very low power requirements of the order of fractions of a milliAmpere (mA). In fact, the cost of sampling a sensor is largely due to the cost of operating the microcontroller and its Analog-to-Digital Converter (ADC), which is of the order of a few milliAmperes (mA). In order to sample an analog sensor (most sensors used today fall in this category), a microcontroller has to operate its ADC to access and convert the sensor's output voltage into discrete numbers. This is a relatively power hungry operation and also typically requires some time to complete since the speed of sampling is bounded by delays such as time required to initialize the ADC, waiting for input line voltages to stabilize and taking multiple samples to ensure correctness of reading. These delays are inevitable due to physical limitations of the microcontroller and sensor and the fact that the sensors themselves are extremely simple electrical devices such as resistors or diodes. Therefore, the energy required to take a sensor reading has remained more or less the same whereas the energy required for transmitting a reading over the network has decreased drastically. With more advances in radio technology we expect this gap to widen further in the future. Hence, the tables are now being turned and one is faced with situations where the cost of sensing can actually outweigh network cost by a significant margin.

Re-examining Sensor Sampling Strategies

In light of the above discussion, we feel there is a need to take a fresh look at sensor sampling and acquisition strategies. We note that even though in case of the MICA2, the sensing cost has a higher contribution to the total energy cost as compared to network cost, which may not be significant enough to outweigh the total network cost of communicating over a multi-hop network. On the other hand, for newer hardware such as the Atmel RCB, the difference is

significant enough to warrant a fresh look at query processing strategies for in-network execution of queries.

Currently almost all query execution plans use the Push approach. Push requires nodes to autonomously sample their sensors and push their readings into the network. Optimizations have been suggested where nodes can avoid sampling their sensors if the result of the query can be deduced from already existing information. These optimizations either involve the use of partial aggregate information or utilize models to determine which sensors to sample. The former approach used in TAG [22] only avoids sampling of sensors directly connected to a node, and does not control sensor sampling of other nodes below it in the aggregation tree. The latter approach used by Deshpande et al. in BBQ [8], makes the decision of which sensors to sample based on data requirements of the model, rather than energy consumption. Deshpande et al. do consider the acquisition cost of sampling a sensor, however, they are primarily interested in utilizing this data to find another sensor which is less expensive to sample based on correlations between various phenomena (example, temperature and battery voltage). Therefore, their approach does not look at the question of whether to entirely avoid sampling a sensor based on energy considerations; rather it depends on a model to determine which sensors to sample and attempts to find replacement sensors if possible, having lower energy cost of sampling.

The Pull approach requires nodes to wait for an explicit command before sampling its sensors. The pull approach has been utilized very frugally in sensor network query processing till now, mainly due to the assumption that network cost is more than the cost of sensing. But both the push and pull approaches have their advantages and drawbacks. The push approach allows nodes to execute their task autonomously and has low latency of query response. However, the push approach also almost always requires a node to sample its sensors barring certain

intermediate nodes as mentioned before. This approach worked well in the past when the cost of sampling a sensor was a minor contributor to the total energy cost. But this may not work that well now since acquiring readings from a sensor makes up the largest fraction of a node's energy consumption by a wide margin. Hence, even if a sensor's reading is not transmitted over the network, simply sampling it causes the node's energy consumption to be quite significant. The pull approach on the other hand only requires nodes to sample their sensors when explicitly asked to. However, using pull naively by pulling readings from sensors in parallel will not prove useful in terms of energy consumption due to obvious reasons. On the other hands, if sensor readings are pulled in an optimal order based on their acquisition costs, this can lead to lower energy costs since sensors will be sampled only when required. Madden et al. [23] have proposed such a technique for TinyDB, however their mechanism is only meant for sensors connected to the same node and provides a locally optimal ordering of sampling of sensors connected to the same node rather than a network-wide ordering of all sensors involved in the query. The disadvantage of the pull approach is that it suffers from higher network cost as it requires the exchange of two messages as opposed to one message in pull. Moreover, its data delivery latency is also higher than push. Hence, we feel that a third plan option namely, a hybrid Pull-Push approach is required when data is pushed from some sensors and pulled from others. In this hybrid approach pull mechanisms are utilized to cut down on unnecessary sampling of sensors thereby reducing energy consumption and push mechanisms are utilized to reduce network traffic and latency of query response. The following sections provide a detailed discussion of all three approaches and describe how a query optimizer generates and chooses the query execution plan which best meets the goal of minimizing node energy consumption while ensuring that the latency of query response does not go beyond user-specified bounds.

Sensor Querying Strategies

We first begin by identifying the type of queries we are targeting for optimization and define some terms which will be used in subsequent sections. Then we provide a description of how a query is disseminated among the nodes for in-network execution based on network topology. Then we cover three querying strategies namely, Push, Pull and hybrid Pull-Push. For each of the approaches we also define cost functions which will be used by the query optimizer to identify the best query plan. Then, we describe how the query optimizer generates and chooses the best query plan for minimizing energy consumption while ensuring that the response latency remains within user-specified bounds. Finally, we describe how the cost performance of a plan can be monitored during its execution to ensure its effectiveness.

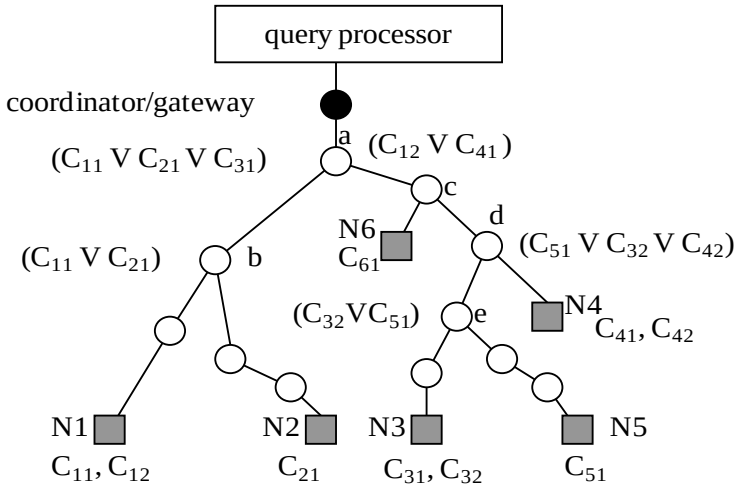

Figure 5-2. Query dissemination and evaluation

Definitions

We focus on continuous selection queries involving multiple predicates applied on multiple sensors, since these types of queries are directly affected by sensor acquisition costs and the order of sampling sensors. We denote these predicate queries in conjunctive normal form or CNF (Equation 5-1). If we consider any query expressed as a SQL statement such as SELECT

<sensor-list> FROM <sensors-list> WHERE <where-clause-expression>, then Equation 5-1 corresponds to the where-clause-expression.

$$Q = \bigwedge_i \left(\bigvee_j C_{ij} \right) \quad (5\text{-}1)$$

Each literal C_{ij} involves a predicate applied on a sensor and is of the form P(s), where P is a predicate and P(s) denotes that predicate P is being applied on sensor 's'. $Sel_P(s)$ denotes the selectivity of sensor 's' with respect to predicate P.

We assume $Sel_P(s) \in [0, 1]$ and is the probability that P(s) evaluates to False. Hence, the higher the value of $Sel_P(s)$, the more selective the sensor 's' is with respect to predicate P. $Sel_P(s)$ can be calculated by the query processor based on sensor history [5] or can be obtained directly from statistics stored onboard the sensor nodes. If the predicate being referred to in the text is unambiguous, we use Sel(s) instead of $Sel_P(s)$. We denote the cost of sampling a sensor 's' as E(s) where E(s) denotes energy consumed by the microcontroller for sampling sensor 's' and is measured in milliJoules (mJ). Note that the microcontroller may consume different amounts of energy based on the type of sensor being sampled. Assume that the node evaluating C_{ij} only transmits the result when C_{ij} evaluates to True.

We define the network cost of transmitting the result of a literal C_{ij} as $NwkCost(C_{ij})$, where $NwkCost(C_{ij})$ is the total energy consumed (mJ) by nodes in the network for transmitting the result to its destination. This cost is the sum total of the energy spent by each node along the route (including the source and destination nodes) for receiving and transmitting the result. NwkCost can be calculated if the network structure and link quality information is available. In a ZigBee network, this information is readily available from the Coordinator (one of whose roles is to maintain an overall view of the network) and the query processor does not need to expend extra effort towards gathering this information from all the nodes in the network. This information is automatically transmitted to the Coordinator by piggybacking it on status update messages.

Query Dissemination

Users issue queries to the query processor residing in the service layer. The query processor generates a suitable query plan after utilizing its optimizer and injects it into the network via the coordinator/gateway. The query is disseminated into the network (Figure 5-3) for execution using an overlay tree structure similar to what is used in TinyDB [23]. We will use this example to illustrate all the three strategies namely, Push, Selective Pull and Pull-Push. In particular example, the query is given by Equation 5-2.

$$Q = (C_{11} \vee C_{21} \vee C_{31}) \wedge (C_{12} \vee C_{41}) \wedge (C_{51} \vee C_{32} \vee C_{42}) \wedge (C_{61}) \quad (5\text{-}2)$$

Evaluation of each C_{ij} is assigned to the node connected to the sensor associated with the literal. For example, C_{11} and C_{12} are pushed to node N1. Hence, a node may have to perform evaluation of literals belonging to multiple clauses.

The cost of evaluating the set of literals assigned to a node depends on the selectivity of each sensor involved and its cost of sampling. Since a node is required to evaluate multiple disjunctions of literals (often belonging to different clauses), the order of sensor sampling is done as follows. For each group of literals assigned to a node belonging to the same clause, the node samples the sensors serially in the ascending order of their selectivity. This ensures that the sensors which are most likely to satisfy their predicates and hence, cause the disjunction to evaluate to True are sampled first. The sampling halts as soon as one of the literals evaluates to True. If we arrange the sensors in ascending order of their selectivity and enumerate them and their associated predicates, then the expected energy cost of evaluating a group of literals assigned to a node belonging to the same clause is given by Equation 5-3.

$$C\left(\bigvee_{i=1} P_i(s_i)\right) = \sum_{i=1}\left[E(s_i)\prod_{k=1}^{i-1} Sel_{P_{k-1}}(s_{k-1})\right] \quad (5\text{-}3)$$

$Sel_{P_0}(s_0)$ is constant and set equal to 1. A node evaluates groups of literals in ascending order of their expected energy cost. Note that for the sake of simplicity, this expression assumes

that each group of literals is evaluated independently. In reality however, if a sensor belongs to more than one group of literals, it is only sampled once and its reading shared among all the literal evaluations. This reduces energy consumption since the Analog-to-Digital Convertor is only turned on once for sampling the sensor as opposed to multiple times for each literal.

Push Strategy

The Push approach is the most widely adopted strategy for sampling sensors and evaluating queries. We cover this approach for the sake of reference and for deriving cost functions, since we will be using certain aspects of Push-based querying in our proposed query plans. During each epoch of execution, nodes sample their respective sensors and create partial evaluation records which they transmit to their parents. Since each node can be assigned literals belonging to multiple clauses, multiple partial evaluation records can originate from a single node. This entire process of evaluation uses the slotted approach of time scheduling as described in [22], where each node divides its epoch into multiple slots and requires its children to respond within a certain sub-interval. Clauses are evaluated in full at the intermediate node serving as the root of the sub-tree containing all the nodes involved in the clause. For example, in Figure 5-2 the final evaluation of the clause $(C_{51} \vee C_{32} \vee C_{42})$ takes place at node 'd'. The conjunction of clauses also gets evaluated in a similar manner to give the final result of the query. The roots of all the evaluation sub-trees get selected during the query dissemination phase, based on routing of clauses and literals down the cluster-tree network. If a node detects that a clause gets split up among its children, then it knows that it is the root of the evaluation tree for that clause. Similarly if a node detects that two or more clauses get routed to different child nodes then it determines it is a root of the evaluation tree for conjunction of those clauses. The intermediate routing nodes are capable of suppressing the transfer of partial evaluation records if they are able to deduce the

final result based on information contained in them. For example, a clause evaluation root node will not transmit the result if it finds out that the clause evaluated to False, since that will result in the entire query which is a conjunction of clauses to evaluate to False.

The energy cost of evaluating a clause using Push strategy can be calculated by modifying the expression given by Equation 5-3 to factor in the cost of evaluating literals and associated network cost for all nodes involved in the clause (denoted by the set *Node*). The expected evaluation cost of a clause $\underset{N \in Node}{\vee}(\underset{i=1}{\vee} P^{N}{}_{i}(s^{N}{}_{i}))$ using push approach (denoted by $EvalCost_{Push}$) is given by Equation 5-4.

$$\mathrm{C}\left(\underset{N \in Node}{\vee}(\underset{i=1}{\vee} P^{N}{}_{i}(s^{N}{}_{i}))\right) = \sum_{N \in Node}\left[(1-\prod_{i=1} Sel_{P^{N}{}_{i}}(s^{N}{}_{i}))NwkCost(C_{N}) + \mathrm{C}(\underset{i=1}{\vee} \mathrm{P}^{\mathrm{N}}{}_{\mathrm{i}}(\mathrm{s}^{\mathrm{N}}{}_{\mathrm{i}}))\right] \quad (5\text{-}4)$$

N is the Id of a node belonging to the set *Node*, $Sel_{P^{N}{}_{i}}(s^{N}{}_{0}) = 1, \forall i$, $C_{N} = (\underset{i=1}{\vee} P^{N}{}_{i}(s^{N}{}_{i}))$ and partial clause evaluation cost $\mathrm{C}(\underset{i=1}{\vee} P^{N}{}_{i}(s^{N}{}_{i}))$ is given by Equation 5-3. The sum inside the box brackets gives the total energy consumed by a node for evaluating the literals and transmitting the result over the network if any of the literals evaluates to True. The final summation is over all members of the *Node* set.

$$\mathrm{TotalCost}_{\mathrm{Push}}(\underset{i}{\wedge}(\underset{j}{\vee} C_{ij})) = FinalAggregationCost + \sum_{i} EvalCost_{Push}(\underset{i}{\vee} C_{ij}) \quad (5\text{-}5)$$

In Equation 5-5, *FinalAggregationCost* is the additional network cost required to aggregate partial results created during evaluation of each clause and aggregation of results from multiple clauses to obtain the final result of the conjunction. The NwkCost term is calculated based on the number of hops required to transmit the partial evaluation record to the next intermediate node which will merge it. For example, referring to Figure 5-2, $\mathrm{NwkCost}(C_{11}) \propto 2$ (the number of hops between nodes N1 and 'b') whereas $\mathrm{NwkCost}(C_{12}) \propto 3$ (the number of hops between nodes N1 and 'a'). In case the evaluation tree structure is such that the final result can be obtained as a side-effect of clause evaluation then *FinalAggregationCost* will be minimal.

Finally, the total energy cost for in-network evaluation of a query ($Q = \wedge_i(\vee_j C_{ij})$) using push strategy is given by Equation 5-5. This total energy cost is used by the query optimizer for comparing different plan options to decide on the most optimal plan for distributed execution of queries injected by the user.

Selective Pull Strategy

The order of evaluation in the Push strategy is governed by the structure of the evaluation tree which in turn depends on the topology of the network. The naïve Pull strategy would involve pulling all the sensor readings in parallel and essentially following the same evaluation order as above, but such a plan would be of no use since it will always have higher energy cost due to extra network traffic and double the latency of the Push strategy.

Instead of the naïve approach, we propose a selective pull strategy where the sampling of sensors across the network is ordered according to the expected cost of evaluating of clauses involving them and clauses are evaluated in serial order. Consider the example depicted in Figure 5-2. For the sake of discussion, suppose the optimal order of evaluation obtained by the query optimizer (based on ascending cost of evaluation) is:

$$(C_{11} \vee C_{21} \vee C_{31}) \rightarrow (C_{51} \vee C_{32} \vee C_{42}) \rightarrow (C_{61}) \rightarrow (C_{12} \vee C_{41})$$

The first step is the query dissemination phase has been described before, where the entire query is pushed on the network. It has been suggested that for the pull approach, nodes do not need to store query information [29]. However, we feel that for continuous queries it is more energy efficient and fault tolerant for nodes to store the query instructions even though they do not execute them without the arrival of an explicit command from a parent node.

The next step involves evaluation of the first clause in the execution schedule. In the above example node ‘a’ which is the root of the 1st clause’s evaluation tree transmits a pull command to

nodes ‘b’ and ‘c’. These nodes in turn relay the command to nodes N1, N2 and N3. The root does not issue 3 pull commands rather the number of pull commands issued is equal to the number of its children it needs to transmit the command to (in this case, 2). After each of the nodes receive their respective pull commands, they sample their sensors in the order of their sampling cost and selectivity.

One important thing to note here is that a pull command does not reference a particular sensor. In fact, when a node receives a pull command it samples sensors associated with all the literals assigned to it. Recall that a node can have multiple sensors spread across multiple clauses. Hence, by adopting this strategy a node does not have to sample its sensors multiple times. This naturally saves energy consumption due to networking but it also saves a significant amount of energy consumed by sensor sampling. This is due to the fact that it is far more energy-efficient to initialize the ADC once and sample all the sensors rather than keep starting and stopping it for each individual sensor sampling. Hence, when a node sends a response to a pull command it transmits multiple partial evaluation records up to the root.

Once a node finishes evaluation, it transmits the partial records to the root of the evaluation tree. The partial records get merged on their way up to the root and once the root receives all the responses and it is able to determine whether the clause evaluated to True or False.

If it evaluated to False, the execution of the query for that epoch is terminated since it results in the entire conjunction expression evaluation to False. If the clause evaluates to True, the root of this clause’s evaluation tree (node ‘a’ in the example) transmits the partial evaluation records of the other clauses to the root of the next clause’s evaluation tree (node ‘d’ in the example). This root in turn examines the partial evaluation records and sends pull commands to only those nodes which have sensors whose readings are not in any of the partial record. The

process of execution continues in this manner till one of the clauses evaluates to False or all clauses evaluate to True, in which case a response is sent to the query processor.

We can observe that the effectiveness of the selective pull strategy depends on the order in which clauses are evaluated across the network. Ordering of sensor samples with the aim of minimizing acquisition costs has been studied before by Madden et al. in [23] but their mechanism only provides a locally optimal ordering of sampling of sensors connected to the same node rather than a network-wide ordering of all sensors involved in the query.

The cost effectiveness of the selective pull strategy depends not only on the cost of evaluating a clause but also on the cost of transferring control from the root of one clause evaluation tree to the next. The formula for calculating the cost of evaluating a clause using pull is similar to $EvalCost_{Push}$ described before except it contains an additional term corresponding to the network cost of the pull command. The expected evaluation cost of a clause using pull approach is given by Equation 5-6.

$$EvalCost_{Pull}((\bigvee_{N \in Node}(\bigvee_{i=1} P^{N}{}_{i}(s^{N}{}_{i}))) = \sum_{N \in Node}\left[NwkCost(C_N)\right] + EvalCost_{Push} \quad (5\text{-}6)$$

N is the Id of a node belonging to the set *Node* and $C_N = (\bigvee_{i=1} P^{N}{}_{i}(s^{N}{}_{i}))$.

The cost of transferring control from the root of one evaluation tree to the next is basically the network transmission cost for transferring the partial records to the other root. Both these pieces of information can be easily obtained by the query processor. Given an ordered pair of two clauses, the service layer can provide information as to the possible number of partial evaluation records that need to be transferred based on which sensors are connected to which nodes. Furthermore, based on network information obtained from the ZigBee coordinator/gateway the query processor can also find out the cost of transmitting those records from one root node to the other.

The total expected energy cost of evaluating a query ($Q = \wedge_i (\vee_j C_{ij})$) using pull strategy (where clauses are numbered in ascending order of execution), is given by Equation 5-7.

$$\text{TotalCost}_{\text{Pull}}(\wedge_i(\vee_j C_{ij})) = \sum_i p_i (TrfCost_{(i-1)\rightarrow i} + EvalCost_{Pull}(\vee_i C_{ij})) \quad (5\text{-}7)$$

$TrfCost_{(i-1)->i}$ denotes the network energy cost of transferring control from root of evaluation tree of clause numbered 'i-1' to the root of the evaluation tree of clause numbered 'i'. p_i denotes the probability that clause 'i' will be executed, that is, it is the probability that execution control will transition from clause 'i-1' to 'i'.

This probability is dependent on the probability that the previous clause 'i-1' evaluates to True. $p_1 = 1$ and $p_i = \prod_{k=1}^{i-1} p'_k$, (for $i \geq 2$), where $p'_k = P(\vee_j C_{kj} = True)$. This implies that $p'_k = 1 - P(\vee_j C_{kj} = False)$ which further implies that $p'_k = 1 - P(C_{kj} = False \forall j)$. Hence, $p'_k = 1 - \prod_{s \in A_{kj}} Sel(s)$, where A_{kj} is the set of sensors involved in evaluation of clause $\vee_j C_{kj}$. Note that there is an assumption of independence of sensor selectivity in the formula for p'_k. If possible one can refine this probability by using correlation information about various phenomena such as the approach used by Deshpande et al. [8].

As discussed previously, determining the optimal order of execution is essential for the selective pull approach to succeed. The goal here is to determine the optimal order of evaluating clauses so that the total energy consumption is minimized. Consider a query $Q = \wedge_i B_i$. We can view the execution space as a complete directed-graph G = (V, E) where clause B_i in the query Q is represented by vertex V_i. The weight of the edge E (i, j) going from a vertex V_i to V_j is set equal to $p'_j (TrfCost_{i->j} + \text{EvalCost}_{\text{Pull}}(Bj))$ where p'_j, $EvalCost_{Pull}(B_j)$ and $TrfCost_{i\rightarrow j}$ are defined above. The edge weight gives the expected cost of transitioning from V_i and V_j based on transition probability $p'i$. Also, a dummy node D is added to G such that E (D, i) is equal to

$EvalCost_{Pull}(B_i)$ and E (i, D) = 0, with 'i' ranging from 1 to |V|. Thus, the problem of optimizing the order of clause evaluation with the goal of minimizing total energy consumption can now be solved as a Traveling Salesman Problem (TSP) with starting point as node D. There have been numerous solutions proposed for the Traveling Salesman Problem such as dynamic programming for finding optimal solutions and heuristic techniques such as simulated annealing and local search for near-optimal solutions. Since the dynamic programming solutions have time complexity which is exponential in terms of the number of clauses being evaluated, it is more practical to use one of the heuristic techniques in the query optimizer.

Hybrid Pull-Push Strategy

Both the push and selective pull strategies have their advantages and disadvantages. The push strategy provides greater autonomy to the nodes and works without external supervision. It also will typically have lower latency of query response as compared to selective pull. The selective pull strategy on the other hand, takes into consideration the fact that sensing costs now significantly dominate the total energy consumption and tries to optimize the order of sensor sampling across the network in order to minimize energy consumption. However, this comes at a cost of greater reliance on the network and higher latency as compared to the push approach, since each node is subject to external supervision from a parent node.

We feel that one of the best ways to utilize the strong points of both push and pull strategies is to adopt a hybrid pull-push approach. The main goal of the hybrid pull-push approach is to minimize the total energy required to execute a query in the network while ensuring that the latency of query response is within bounds specified by the user. In order to achieve this, the plan utilizes the push approach on a number of sensors based on cost and latency considerations and utilizes the selective pull approach on the rest of the sensors. After query dissemination, the group of sensors corresponding to a set of clauses having the lowest

evaluation cost is asked to execute the push strategy. Since the query is a conjunction of clauses therefore, only if all the initially selected clauses evaluate to True, are the other clauses evaluated using the selective pull approach. The construction of a hybrid pull-push plan is described in more detail when we discuss how the query optimizer chooses the best plan of execution.

The energy cost of a hybrid pull-push plan depends on which of the clauses were evaluated using push approach and which were evaluated using selective pull. The total energy cost is simply the sum of the energy cost of each clause calculated by applying the appropriate cost formula for the push or pull approach. Suppose the query in question in $Q = \bigwedge_{i=1}^{n} B_i$ where the clauses are numbered in the order of execution. Suppose the hybrid plan calls for 'm' clauses to be evaluated using push and the rest evaluated using pull, thereby implying in this case that the first 'm' clauses will be evaluated using push. The total energy cost of the hybrid pull-push plan is given by Equation 5-8.

$$\text{TotalCost}_{\text{Hybrid}}\left(\bigwedge_{i=1}^{n} B_i\right) = \sum_{i=1}^{m} EvalCost_{Push}(B_i) + \sum_{i=m+1}^{n} p_i\left(TrfCost_{(i-1)\rightarrow i} + EvalCost_{Pull}(B_i)\right) \quad (5\text{-}8)$$

Choosing the Best Query Plan

Heidemann et al. [12] proposed that data dissemination algorithms need to be mapped to application requirements. Through our hands-on experience we found that the actual fulfillment of these application requirements eventually depends on the specific sensor hardware that is being targeted. Hence, the query optimizer needs to be flexible enough so that it can generate multiple query plans which exploit specific characteristics of different hardware with the aim of satisfying application requirements without attempting to generate a "one-size-fits all" solution. The *Sensable* query optimizer generates all three types of query plans discussed in the preceding sub-sections and chooses the one which best minimizes the energy consumption while meeting

the user requirements on latency of query response. To aid the query optimizer in making a decision a user is required to provide information regarding its tolerance for latency. Instead of requiring the user to provide a hard number, the query optimizer asks for tolerances in terms of percentage (denoted by L_{max}). The percentage value represents the magnitude by how much the latency can exceed that of the Push approach. If L_{Push} and L_{Plan} respectively denote the latencies for push approach and the plan that was chosen then, $\frac{L_{Plan} - L_{Push}}{L_{Push}} x100 \leq L_{\max}$ must hold true.

If we consider a query Q = $\wedge_i B_i$ then the latency for evaluating it using the push approach is given by Equation 5-9. The latency for the selective pull approach is given by Equation 5-10. The latency of a hybrid plan is calculated as the sum of latencies due to its push and pull operations. If we consider the hybrid pull-push plan then its query response latency is given by Equation 5-11.

$$L_{Push} = Latency_{Push} FinalAggregation + \sum_i Latency_{Push}(B_i) \quad (5\text{-}9)$$

$Latency_{Push}(B_i)$ denotes the latency in receiving a response for clause B_i if it evaluates to True, and $Latency_{Push}FinalAggregation$ denotes the latency in generating the final results from results of the clause evaluations. For the sake of simplicity, these individual latency values can be simply calculated as the total number of hops between the nodes evaluating B_i and the root of B_i's evaluation tree.

Given latency tolerance L_{max}, the query optimizer undertakes the following steps to determine the optimum query plan. First it generates a query plan using the push strategy and calculates the energy cost ($TotalCost_{Push}$) and latency (L_{Push}). Next it generates a query plan using the selective pull approach by solving the optimization problem using simulated annealing and calculates the energy cost ($TotalCost_{Pull}$) and latency (L_{Pull}). Finally, it generates a hybrid pull-push query plan as follows. Suppose the selective pull query plan generated for user-defined query Q = $\wedge_i B_i$ resulted in a query plan with the following order of evaluation of clauses: B_1, B_2,

B_3, ... B_n. The query optimizer progressively replaces the pull action associated with each clause with push, starting from the first clause. Each time it does this, it recalculates the latency of the plan (L_{Hybrid}) and checks if its percentage relative difference with L_{Push} is less than L_{max} or not. This process is continued till the relative difference in latency becomes less than L_{max}. If the final hybrid plan consists of 'm' push based evaluations followed by (n-m) selective pull-based evaluations then there exists no k < m for which Equation 5-12 holds true.

$$L_{Pull} = \sum_i p_i \left(Latency_{Pull}(B_i) + TrfLatency_{(i-1)\rightarrow i} \right) \quad (5\text{-}10)$$

$Latency_{Pull}(B_i)$ is calculated as twice the total number of hops between the nodes evaluating B_i and the root of B_i's evaluation tree. p_i is as defined before. $TrfLatency_{(i-1)->i}$ is the latency in transferring control from the root of clause (i-1)'s evaluation tree to clause i's evaluation tree.

$$L_{Hybrid} = Latency_{Push} FinalAggregation + \sum_{i=1}^{m} Latency_{Push}(B_i) + \sum_{i=m+1} p_i \left(Latency_{Pull}(B_i) + TrfLatency_{(i-1)\rightarrow i} \right) \quad (5\text{-}11)$$

$$Latency_{Push} FinalAggregation + \sum_{i=1}^{k} Latency_{Push}(B_i) + \sum_{i=k+1} p_i \left(Latency_{Pull}(B_i) + TrfLatency_{(i-1)\rightarrow i} \right) \le L_{\max} \quad (5\text{-}12)$$

Finally, the query optimizer compares the energy costs of all the plans which meet the user's latency bounds and chooses the one with the lowest cost. Typically one would expect that the hybrid pull-push plan will be always chosen. However, this may not be true in every case since the cost-effectiveness of the plan depends heavily on energy consumption characteristics of the sensors and the sensor platform hardware. Hence, the approach outlined above allows the optimizer to fall back on traditional approaches such as Push, if the new proposed strategies do no prove to be more cost-effective.

Monitoring Plan Performance

Since the query plan is generated based on a snapshot of history its effectiveness in minimizing energy consumption can decrease over time due to changing conditions in the

deployment environment. In order to monitor the effectiveness of the query plan being executed, each node can keep track of selectivity of its sensors and calculate the expected cost of evaluation of literals assigned to it. This information can be periodically transmitted up the evaluation tree as a partial record getting merged along the way, in the same manner as the query evaluation records. This will lead to the root of each clause's evaluation tree to have updated cost estimates for that clause. For selective pull and pull-push approaches, the total cost also depends on the probability of transferring control from one root to another. Hence, each evaluation tree root also factors into the cost calculation, the expected network cost based on the actual transition probability of transferring control to the next root in the execution sequence. Finally, the query processor in the service layer compares the actual updated cost with the estimated cost and if required, regenerates plans and chooses the best one using more up-to-date selectivity data. The new plan can either be disseminated in its entirety into the network or the query processor only disseminates those portions of the new plan which differ from the existing plan. Selectively adding or removing query execution assignments from specific nodes in this manner, without having to reset the entire query execution process, leads to greater energy efficiency and higher system availability.

Fault Tolerance

In this sub-section, we discuss some mechanisms for dealing with node and network malfunctions, since failure is an integral part of any sensor network. The query execution strategies described utilize in-network evaluation trees and hence, are vulnerable to failure of any of the nodes participating in the evaluation process or nodes which are not participating in query evaluation but are nonetheless playing a vital role by linking up different segments of an evaluation tree. The following are possible types of failure that can affect the query execution plans that are executed in-network:

1. Node whose sensors are involved in evaluation of a clause fail: In such a case, the ancestor of this node responsible for merging its evaluation record will not receive any messages from it. If the node is using the push approach, there are a number of techniques that have been suggested to cope with failure such as the use of child caches [22] and Bayesian techniques to determine whether the lack of response indicates suppression or node failure [30]. In case the node was using the pull approach then its failure will get detected by the next epoch of execution when a pull command is issued. This is due to the fact that ZigBee aims at providing reliable message delivery through acknowledgement mechanisms present in the link layer and above. Hence, a sender is able to determine if its message got delivered or not. The effect of failure of sensors participating in a query depends on the type of query that was issued. For queries targeting specific sensors, the failure of one or more of those sensors may result in the query execution being halted, whereas queries targeting the entire network may not be affected drastically by such failures.

2. Non-participating intermediate node failure: In case an intermediate router node which is not participating in query evaluation fails, then its child nodes simply re-associate themselves with a new parent in the network. This is a feature of ZigBee's self-healing characteristics which allows the network to cope with loss of a limited number of routers without affecting network connectivity.

3. Root of evaluation tree fails: This is the most serious type of failure that can affect query plan execution since the failure of the root will not only cause the evaluation of its clause to fail but might also lead to the entire query execution getting stalled depending upon the location of that node in the network. To handle root failure we utilize a leader election algorithm. Leader election algorithms for ad hoc networks have been previously proposed [24, 33] and we follow a similar philosophy. We propose that for each node which is the root of an evaluation tree or sub-tree, should inform a small set of its immediate neighbors (which are within 1-hop broadcast range) about its status and have them cache all the information necessary to act as a root node. These neighbors are responsible for monitoring whether the root is online or not. Since they are only 1-hop away, the network overhead and network energy cost is not that significant. When one of the nodes discovers that the root in its neighborhood is dead it can take over the role of the root since it already has all the necessary information. To avoid multiple nodes from detecting failure at the same time and becoming roots of the same tree, we can have each node monitor the root's health in round-robin fashion for a certain number of contiguous epochs.

One of the most important practical issues which will occur in the above scenarios is the fact that whenever a node rejoins the network it is assigned a different network address by its new parent. Similarly, when the root of an evaluation tree fails and a new root takes over, naturally the network address of the new root won't match the address of the original root. This can create significant problems since other nodes in the tree will be unaware of such changes and

will be unable to transmit readings or commands. Fortunately, ZigBee provides a solution to deal with such issues in the form of logical addresses. A logical address is an application-specific address independent of the network address which can be user-defined and used for communication between applications. During query dissemination phase, we can assign each node in the tree a unique logical address which it retains for the duration of the query lifetime. Hence, even if a node rejoins the network by associating with a new parent its logical address still remains the same and gets automatically mapped to its new network address. Similarly, when a node takes over as the root of an evaluation tree it can changes its logical address to that of the deceased root. This ensures that it is able to receive evaluation records from the other nodes in the tree without having to inform them of the change.

Experimental Performance Analysis

We compare and analyze each of the query plan options discussed in this paper, based on two performance metrics: energy consumption and latency of query response. We studied the effect of varying sensor selectivity and the numbers of sensors operating in push and pull modes on these two metrics.

Method of Experimentation

Our experiments consisted of simulating the construction and in-network execution of all the three types of query plans discussed in this paper namely, Push, Selective Pull and hybrid Push-Pull. For analyzing the effect of varying the numbers of sensors operating in push and pull modes, the simulator randomly constructed a set of 5 selection queries involving multiple predicates. Each query involved the participation of 16 sensors in the form of range filter predicates. The 16 sensors were generated such that the selectivity of their associated range filters followed the Gaussian distribution with mean 0.66 and variance 0.33. For each iteration, the simulator randomly generated a multi-hop network of sensors nodes (each responsible for the

operation of one or more sensors participating in the query) and generated query plans as described previously. Then, it simulated the execution of each plan using hardware specifications of the Atlas ZigBee node (Table 5-1). For studying the effect of varying mean sensor selectivity, for each iteration the simulator generated a new group of sensors, such that each new group's selectivity followed the Gaussian distribution with mean equivalent to the sensor selectivity for that iteration and variance equivalent to one-third of the selectivity magnitude.

Table 5-1. Atlas ZigBee node hardware specifications

Operation	Current (mA)	Duration (seconds)
Sampling Sensors, Processing or Listening for messages	6.0	2.0
Receive Message @256Kbps	15	0.002
Transmit Message @256 Kbps	15	0.002

Results and Analysis

First we look at the effect of varying the numbers of sensors operating in push and pull mode, on energy consumption per epoch (Figure 5-3). We observe that the energy consumption (measured in milliJoules per epoch) increases steadily as the ratio of the number of sensors operating in push mode to the number of sensors operating in pull mode increases. If we look at the two extreme cases, we observe that the energy consumption for the conventional all push plan is approximately 2.5 times more than the energy consumption of the selective pull plan.

This is due to the fact that in case of the all push plan, each sensor is sampled independently of others in a parallel manner hence, regardless of whether the sensor's reading satisfies the associated range filter or not, the node incurs the energy cost due to sensor sampling. However, in case of the selective pull plan, sensors are sampled in a linear fashion so that if a sensor's reading fails to satisfy its range filter, all subsequent sensors which were supposed to be read after it are not sampled. For ZigBee based sensor nodes such as the ones used by Atlas, the sensing cost makes up for an overwhelming majority of the total energy consumption (Figure 5-

1). Hence, the all push plan has much higher energy costs than the selective pull and any of the hybrid push-pull plans. We notice that in case of the hybrid push-pull plans, their energy consumption falls somewhere between selective pull and all push plans. The hybrid plans which have more number of sensors operating in push mode have higher energy costs as compared to ones which have comparatively more number of sensors operating in pull mode.

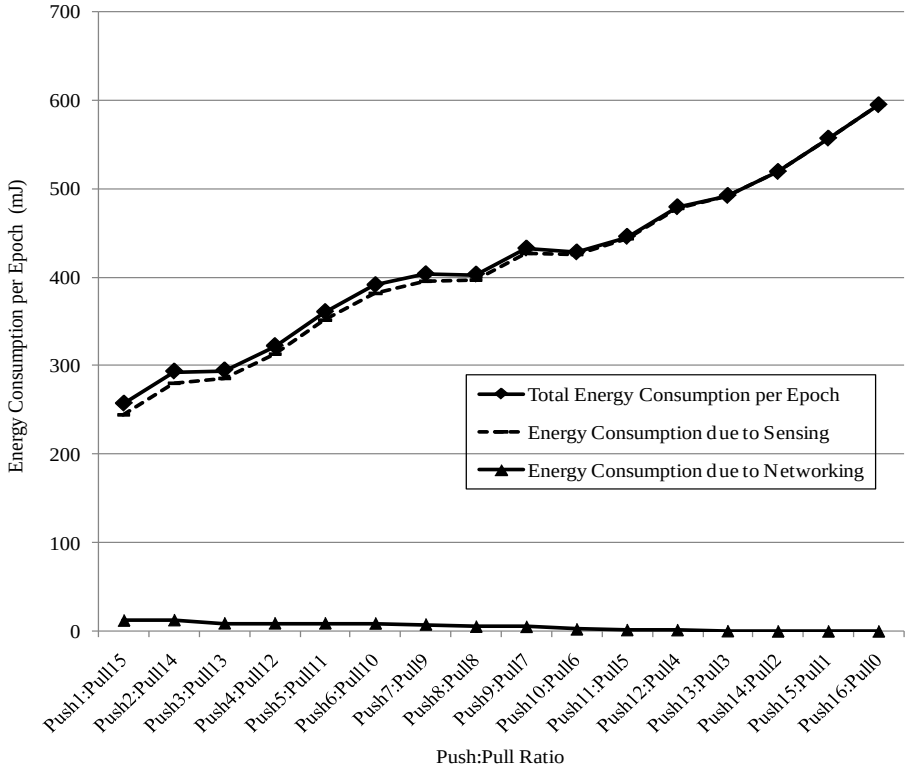

Figure 5-3. Comparing energy consumption of different query plans

However, if we look at the latency of response for the various query plans, we find a different picture altogether (Figure 5-4). We observe that the latency of response decreases as more sensors operate in push mode as compared to pull mode. If we consider the two extreme cases, we find that the average latency in response for the selective pull plan is nearly 20 times more than the all push plan. However, in case of the hybrid push-pull plans as the number of sensors which operate in push mode increases the latency comes down drastically. This is due to the fact all sensors operating in push mode are sampled in parallel whereas sensors operating in

selective pull mode are sampled in serial order with the control of the execution process passing from one clause to another, which results in higher latency. This clearly brings forth the question of a tradeoff where a user has to decide whether energy consumption is of greater concern or the latency of response. In most cases, we feel one of the hybrid push-pull plans will usually satisfy a user's requirements by providing more energy efficiency as compared to the conventional all push plan at the cost of a small increase in latency of response.

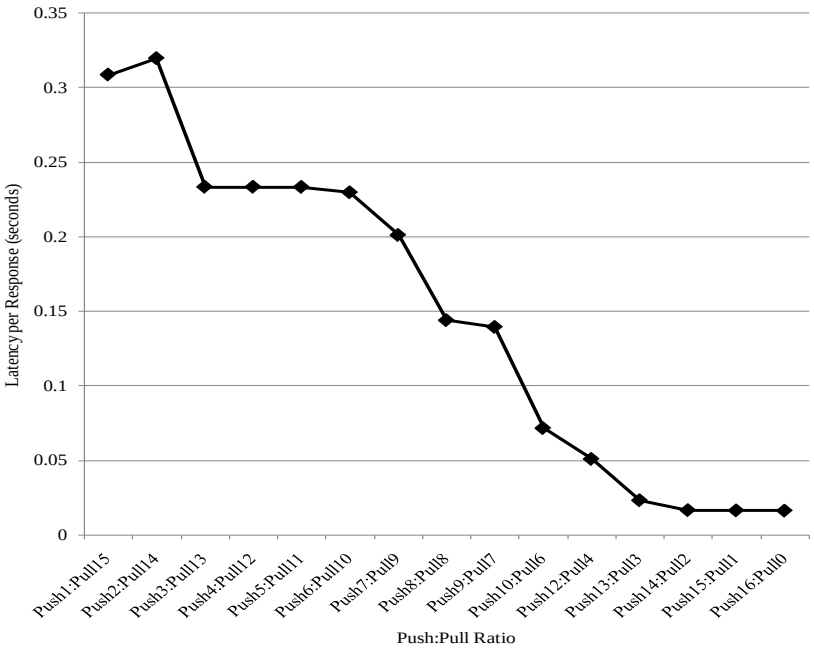

Figure 5-4. Comparing latency of response for different query plans

Next, we look at the effect of varying sensor selectivity on the energy consumption of different query plans (Figure 5-5). We compare the performance of the all push plan with that of the best hybrid query plan that was generated by the query optimizer (in terms of energy consumption per epoch). We observe the energy consumption for the all push plan does not change significantly with the change in average selectivity of sensors. This is due to the fact that in case of push, the selectivity only affects the fact whether a reading is transmitted over the network or not. Since the network cost in case of ZigBee-based hardware is a tiny fraction of the overall energy cost hence, it does not affect the energy consumption significantly.

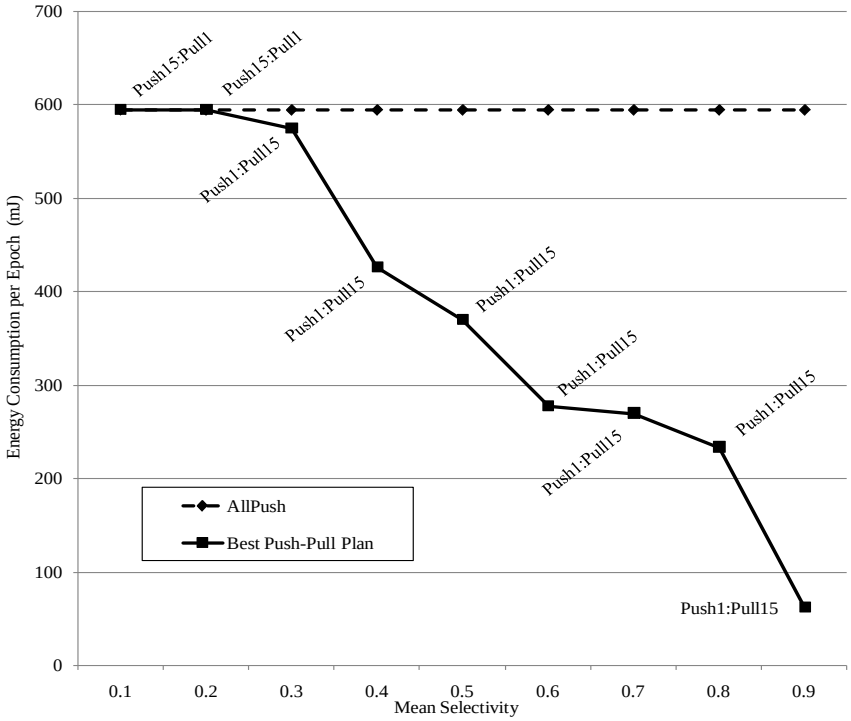

Figure 5-5. Effect of selectivity on energy consumption of query plans

In case of the hybrid push-pull plans we notice that for low sensor selectivity the best plan is usually the one with a high number of sensors operating in push mode and consequently, a very low number of sensors in pull mode. This can be explained by the fact that since average selectivity is low hence the probability a sensor's output satisfies the associated range filter is quite high. Hence, in such a case the selective pull strategy will perform much worse than the push strategy since most of the time almost all the sensors will have to be sampled. The push strategy will be able to handle this in a much more energy efficient way since it does not require sensors to be sampled in serial order and hence, does not incur the extra network cost of hopping from one clause to another. However, as the average selectivity increases the hybrid plans with higher number of sensors operating in pull mode prove to be more energy efficient. Furthermore, the energy consumption of the hybrid plans also drop drastically. This is due to the fact that as selectivity increases the probability that a sensor's output fails to satisfy its associated filter becomes higher. This in turn implies that in case of selective pull, the probability that the

subsequent sensors do not have to be sampled also becomes higher. Since sensing cost is a major portion of the total energy cost hence, the energy saved by not sampling these sensors leads to a significant reduction in total energy consumption. We also notice that for selectivity less than 0.3, the all push plan is the best in terms of energy consumption. However, as soon as selectivity exceeds 0.3, the selective pull plan seems to have the lowest energy drain. The reason for such a dramatic shift from all push to selective pull is due to the fact that when selectivity is less than 0.3, the deciding factor is the network energy cost. Whereas, when selectivity exceeds 0.3, the deciding factor becomes the sensor sampling cost. The cost of sampling one sensor has a much larger magnitude than the cost of transmitting one network packet. Hence, for higher values of selectivity, reducing the probability of sampling even one additional sensor leads to a larger impact on total energy consumption which usually makes selective pull the most energy efficient plan in such cases.

If we consider the effect of varying selectivity on the latency of query plans (Figure 5-6), we observe that the all push plan is consistently better in terms of lower latency as compared to the best hybrid push-pull plan generated by the query optimizer. This is due to the fact that selectivity does not affect latency of the all push plan. However, it does negatively affect the latency of any query plan which has a selective pull component. Since selectivity determines the number of sensors that will eventually get sampled using a selective pull strategy hence, having low selectivity increases the latency of the plan significantly. As average selectivity increases the latency of the hybrid plan decreases somewhat however we note that it stays within the range of 0.03 to 0.04 seconds. This is due to the fact that any gains made due to not sampling some sensors might get offset by the additional latency brought on by having to hop from one clause to another in a serial fashion. We also observe that for extremely high selectivity, the best hybrid

plan is actually the selective pull plan since it takes advantage of the fact that a sensor's output will fail to satisfy its associated range filter most of the time hence, eliminating the need to sample the rest of the sensors.

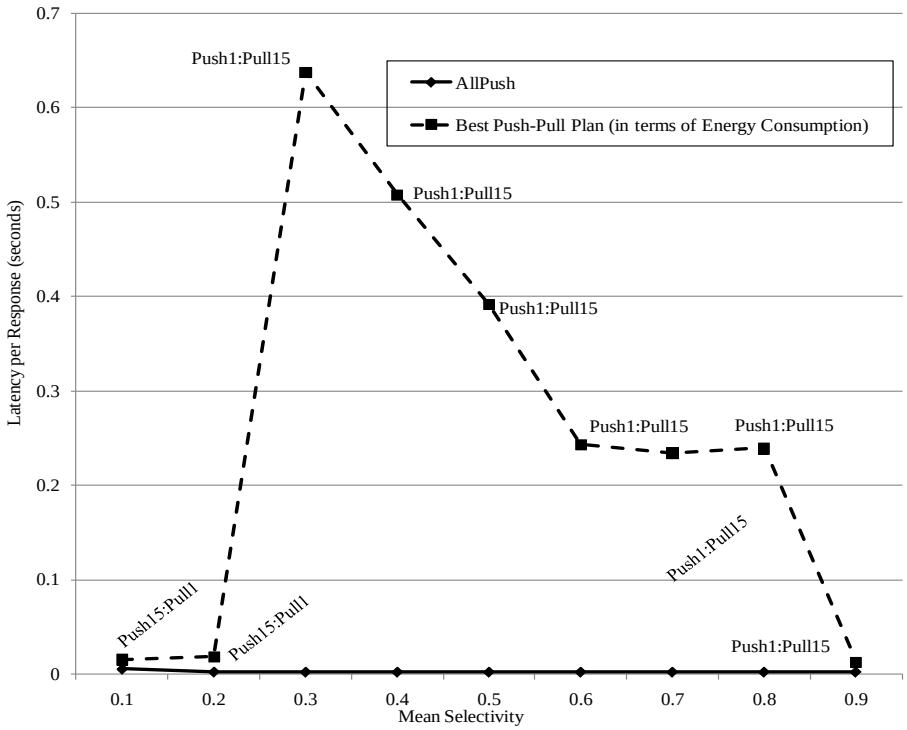

Figure 5-6. Effect of selectivity on latency of query plans

Next, we look at the effect of the varying the number of sensors participating in the query, on energy consumption (Figure 5-7). We observe that the energy cost of the all push query plan increases almost linearly as the number of participating sensors increases. This can be attributed to the fact that since all sensors are sampled in parallel and the sensing cost is by far the largest contributor to the total energy consumption hence; as the number of sensors increase the total energy consumption goes up by nearly the same factor. However, in case of best push-pull query plans we observe that the increase in energy consumption is much more gradual. Moreover, we notice that the relative difference between the energy costs due to the all push plan and the best hybrid plan increases as the number of sensors increase. This is due to the fact that unlike all push strategy, in case of selective pull strategy simply increasing the number of sensors

participating in the query does not necessarily imply that those sensors will be sampled every time the query is executed. In fact, as explained previously the question of how many sensors are sampled during each execution depends on sensor selectivity rather than the total number of sensors participating in the query. Hence, the total energy consumption does not increase by the same factor and we can say that the hybrid plans or selective pull plans scale better in terms of number of sensors participating in the query.

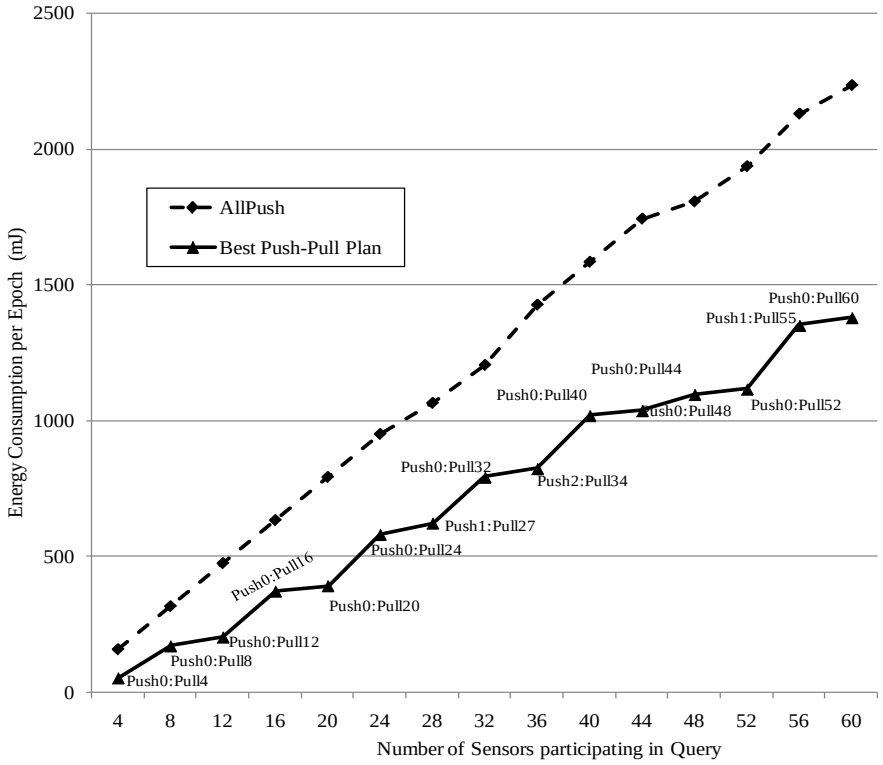

Figure 5-7. Effect of number of sensors on energy consumption of query plans

Finally, we take a look at the effect of varying the number of participating sensors, on the latency of query plans (Figure 5-8). We find that the latency of the all push plan remains more or less uniformly low regardless of the number of participating sensors. This is due to the fact that all sensors in the all push plan are sampled in parallel and their readings pushed and merged up the network. So the effect on latency due to increasing the number of sensors is minimal. On the other hand, for selective pull and hybrid plans, the latency increases as the number of

participating sensors increase. This is due to the fact that in these plans at least some of the sensors are sampled in a serial fashion using selective pull strategy. Hence, as the number of sensors increases, the length of the queue of sensors being sampled increases and so does the latency of response.

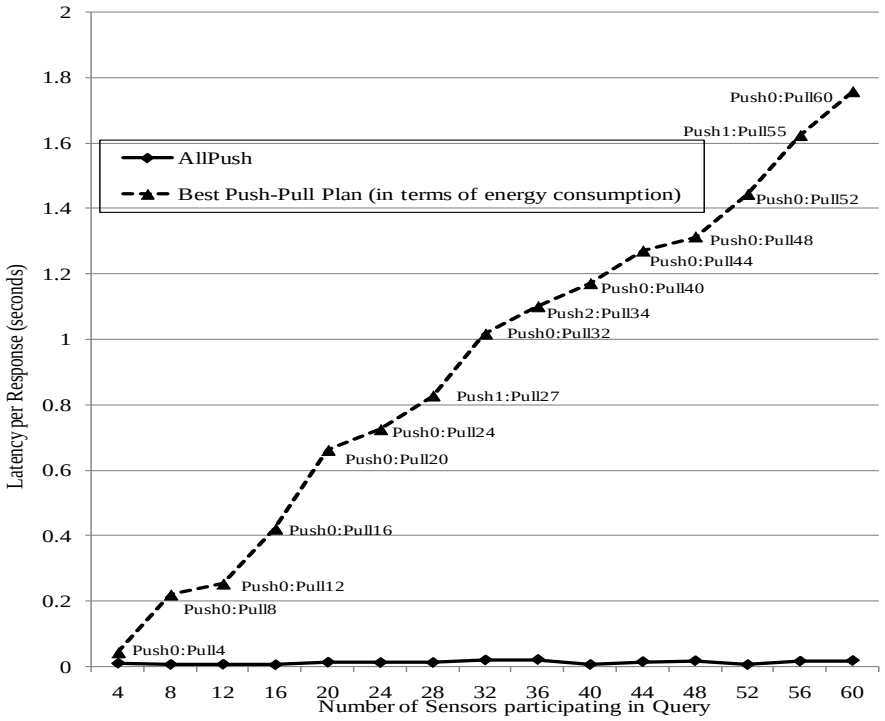

Figure 5-8. Effect of number of sensors on latency of query plans

In conclusion, we can state that in general if the user is only concerned about conserving energy then the selective pull plan will usually be the plan of choice. On the other hand, if fast response time and low latency is the primary concern then the all push plan will be the one desired. However, if a user wants to conserve energy but is willing to tolerate somewhat slower response times then one of the intermediate hybrid push-pull plans might be chosen where the specific ratio of sensors operating in push mode versus pull mode will be determined by the user's preferred tradeoff between energy efficiency and latency. In the end, the choice of query plans is determined by application requirements and hardware capabilities of the deployed sensor network.

CHAPTER 6
VIRTUAL SENSORS FRAMEWORK

Most sensor network database systems are geared toward answering user queries involving raw data, directly originating from individual sensors. However, as sensor network applications become more sophisticated, there is a real requirement for mechanisms which enable users to query data types of more abstract nature and over a larger spatial scope with user-specific reliability requirements.

We define virtual data as a type of data whose values are a function of a variable number of multiple sensory inputs. This can be due to inherent sophistication of the data type itself or the spatial scope within which it is being queried. An example of virtual data would be weather. There is no single sensor which can detect weather since it is an amalgamation of different types of environmental inputs such as temperature, atmospheric pressure and humidity. But if users are only interested in querying weather conditions, it would be inefficient on their part to individually query temperature, pressure and humidity sensors and then aggregate their readings every time the query is executed. Furthermore, user applications may not have access to domain-specific knowledge required to fuse raw sensor inputs into the desired output.

Another example of virtual data is when a user queries for the temperature of a room which has multiple sensors deployed in it. Even though the actual data type is still primitive and can be answered by a single sensor, the query itself involves a specific spatial scope which is larger than each individual sensor. The user is only interested in the temperature of the whole room and not that being reported by each individual sensor. In such a case, it is also more efficient for the query processing system to automatically abstract away the presence of multiple sensors and behave as if there is one large temperature sensor covering the entire room. Putting the onus of aggregating readings over the spatial domain or multiple inputs on the application not only

makes it inefficient and expensive, but also infeasible due to the dynamic nature of sensor networks, where sensors can come online or fail without warning. To address these requirements, we propose a Virtual Sensors Framework, which utilizes SOA mechanisms to provide on-demand creation and life-cycle management of virtual sensors, and process queries involving abstract derived types of data. Virtual sensors are software sensors which do not exist physically; however, they interact with applications as services just like the physical sensor services. Moreover, virtual sensors get deployed on an on-demand basis depending on queries currently executing in the network. Based on queries issued by users, the virtual sensor framework utilizes the service-oriented architecture of SOSNs for dynamic composition of physical sensor services into virtual sensors. As part of the Virtual Sensors Framework, we developed distributed, in-network algorithms for enabling virtual sensors and monitoring their data quality. Furthermore, we also developed distributed fault tolerance mechanisms which compensate for failures of multiple physical member sensors.

The rest of this chapter is organized as follows. We begin by introducing the concept of virtual sensors which enables the querying of virtual data. Then, we classify the different types of virtual sensors according to their roles and discuss the virtual sensor framework model and its components. Next, we describe the on-demand creation of virtual sensors with the help of an example and show how the virtual sensors framework enables a smart space to detect when its sentience gets enhanced. Then, we provide detailed descriptions of basic and derived virtual sensor operations including their distributed, in-network execution; fault tolerance and data quality monitoring algorithms. Finally, we perform experimental analysis of fault-tolerance and data quality monitoring algorithms, and compare the energy efficiency of the virtual sensors framework with that of other contemporary virtual sensor systems.

The Concept and Classification of Virtual Sensors

We define a virtual sensor as a software sensor service entity consisting of a group of physical or other virtual sensors along with associated knowledge which enables query processing involving virtual data. We classify virtual sensors into three categories, namely, *Singleton virtual sensor, Basic virtual sensor* and *Derived virtual sensor*. Each category of virtual sensor is modeled as a set of tuples which represents the knowledge possessed by them.

Singleton Virtual Sensor

This type of virtual sensor represents a single physical sensor. It contains specific knowledge about the sensor in form of attributes such as location, the phenomena it detects, unit of measurement etc. For our purposes we assume that a physical sensor outputs a numeric value which is then converted using a conversion formula into a real-world measurement. The role of the singleton virtual sensor is to enable the creation and composition of other virtual sensor types. We model a singleton virtual sensor as a 5-tuple (Equation 6-1)

$$S = \langle I, L, T, U, F \rangle \quad (6\text{-}1)$$

where,
I: Sensor ID
L: Location of sensor
T: Type of phenomenon detected (temperature, velocity, etc.)
U: Unit of measurement for the phenomena
F: Conversion formula F: $X \rightarrow Y$, where X is the range of numeric readings of the physical sensor and Y is the range of real world measurements.

Basic Virtual Sensor

A basic virtual sensor is composed of a group of singleton virtual sensors of the same type. A basic virtual sensor detects the same type of phenomena which is detected by its member singleton virtual sensors. The role of the basic virtual sensor is to answer queries on virtual data which specify the spatial scope and hence, require the abstraction of multiple sensors of the same type deployed in the space into a single sensor entity. For example, a basic virtual temperature

sensor for a room will be composed of multiple singleton virtual sensors of type temperature which are deployed in that room. Equation 6-2 describes a basic virtual sensor.

$B = \langle I, L, T, U, F, \mathbf{S} \rangle$ (6-2)

where,

S: Set of singleton virtual sensor instances which are members of this basic virtual sensor.

F: Aggregation formula where, $F = f(\{R_s\}_{s \in S})$; R_s is a reading from singleton virtual sensor $s \in S$ and range of F is the set of aggregated readings.

I, L, T and U have their usual meanings defined above.

Derived Virtual Sensor

A derived virtual sensor is composed of a group of basic and/or other derived virtual sensors of heterogeneous types. The role of a derived virtual sensor is to answer queries on virtual data which cannot be answered by individual singleton or basic virtual sensors. This is due to the fact that the type of data required as output is too abstract to be generated by any of the other individual sensors. Consider an example where the application wants to query the weather in a specific location which contains a deployment of multiple temperature, atmospheric pressure and humidity sensors. These physical sensors can be encapsulated as singleton virtual sensors, which in turn, can be grouped by type into basic virtual sensors. Finally, a derived virtual sensor for sensing weather can be composed out of these basic virtual sensors. A derived virtual sensor is modeled as shown in Equation 6-3.

$D = \langle I, L, Y, Z, F, M, R \rangle$ (6-3)

where,

Y: Set of basic/derived virtual sensor types which are required by the derived sensor.

Z: Set of basic/derived virtual sensor instances which are members of the derived sensor. F – Formula for integrating various sensor readings where $F=f(\{R_z\}_{z \in Z})$, where R_z is a reading from basic/derived virtual sensor $z \in Z$.

M: $F \rightarrow R$ where R is the set of possible responses that can be given by this derived virtual sensor.

I and L have their usual meanings.

An interesting feature of a derived virtual sensor is that a large number of such sensors can be created from a small finite set of physical or singleton sensor services. Simply changing the

logical glue which fuses inputs from such sensors in a domain-specific way can provide a rich set of sensing capabilities to the space without any additional hardware deployment cost.

System Framework for Virtual Sensors

We propose a framework model (Figure 6-1) for managing the life cycle of virtual sensors. The framework resides as component of the *Sensable* query processor in the service layer and consists of the knowledge base, framework controller and the virtual sensors currently active in the network.

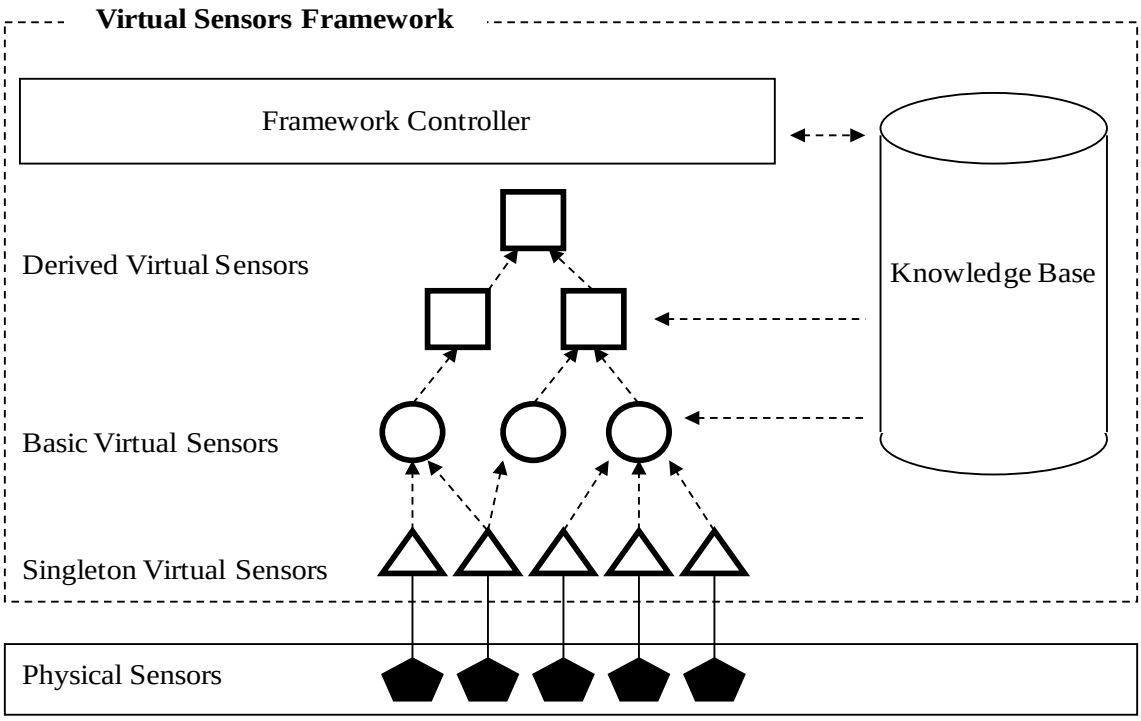

Figure 6-1. Virtual Sensors Framework architecture

The Knowledge Base

The knowledge base is the central repository of information of the framework model. It is responsible for managing the storage, addition, modification and deletion of information records associated with virtual sensors running inside the framework. It stores the following types of

information records which are available for lookup by both the framework controller and active virtual sensors:

1. Sensor Model Definition: These records store the model information for each of the 3 types of virtual sensors defined before: singleton, basic and derived.

2. Virtual Data definition: These records store the definitions of various types of virtual data that can be detected by the sensor network. Examples include basic phenomena such as temperature and humidity and more complex phenomena such as weather, ambience and security. A virtual data definition is stored as a 2-tuple: V = <N, D>; where, N is the identifying name of the phenomenon and D is stored model definition of the derived / basic virtual sensor which can detect this type of data.

Framework Controller

The framework controller is responsible for the overall operation of the Virtual Sensors Framework. It receives virtual data queries from the query processor and then determine with the help of the knowledge base, which virtual sensors need to be created. It is also responsible for removing virtual sensors which are not being used anymore. Furthermore, it also prompts the query processor whenever the sentience of the smart space gets upgraded as a result of introducing a new physical sensor into the space.

We assume that a query from a user application is of the form "SELECT <virtual_data_type> FROM <location> [<Quality_Threshold>]". The query processor extracts the type of virtual data being queried, the location where it has to be detected and the quality threshold and passes it on to the framework controller. The framework controller then starts the process of creating and organizing virtual sensors in the framework to enable the sensor network to work towards fulfilling that query. This process is described in further detail in the following sections. The framework controller also keeps track of sensors joining the network and for each new sensor, it creates a singleton virtual sensor. An important point to note here is the fact that basic and derived virtual sensors are created and destroyed based on user requirements or as a

result of quality control, but singleton virtual sensors are created whenever new physical sensors join the sensor network and are destroyed when the physical sensor they are representing fails or goes offline.

On-Demand Creation of Virtual Sensors

The virtual sensor framework performs dynamic composition of sensor services in response to a user's requirements. It only requires a user to state the type of phenomenon and the location where it is to be sensed at. Based on the type of phenomenon that has to be sensed, the framework controller looks up the corresponding definition from the knowledge base. This definition contains a reference to the sensor model definition of the basic or derived virtual sensor which can detect that phenomenon. The framework controller retrieves the sensor model Definition from the knowledge base and sets some of the attributes in the model definition such as the sensor ID and sensor location. Based on the model definition, it then creates a new service object representing this virtual sensor. This newly created virtual sensor in turn, uses information from its model definition to create or access other service objects representing virtual sensors required by it and the process continues. Once all the virtual sensors have been created and organized, the user is able to retrieve data from the sensor network. In case a virtual sensor fails to start due to a software or hardware error, it returns an error message which is forwarded back up the hierarchy till it reaches the framework controller. The framework controller in turn notifies the query processor that its query cannot be executed at the present time.

Virtual Sensor Composition Graph

To facilitate the on-demand creation of virtual sensors and to enable the smart space to detect when its sensing capabilities are enhanced due to introduction of a new physical sensor into the space, the knowledge base utilizes a sensor composition graph with hash-indexed nodes (Figure 6-2). The sensor composition graph is created from the sensor model definitions stored in

the knowledge base. Each node in the graph represents a singleton, basic or derived virtual sensor and contains its sensor model definition. The sensor composition graph is a directed graph where an edge from sensor A to sensor B indicates that sensor B is a member of sensor A. For example, there exists an edge from the node representing virtual temperature sensor to the node representing virtual ambience sensor (Figure 6-2). Each node in the graph is also hash-indexed with phenomena types being used as keys. This allows constant-time look-ups to initiate the virtual sensor creation process depending on the type of basic or derived phenomenon that has to be sensed, without having to traverse the entire graph. We utilize an example scenario to demonstrate the utility of the sensor composition graph in subsequent sections.

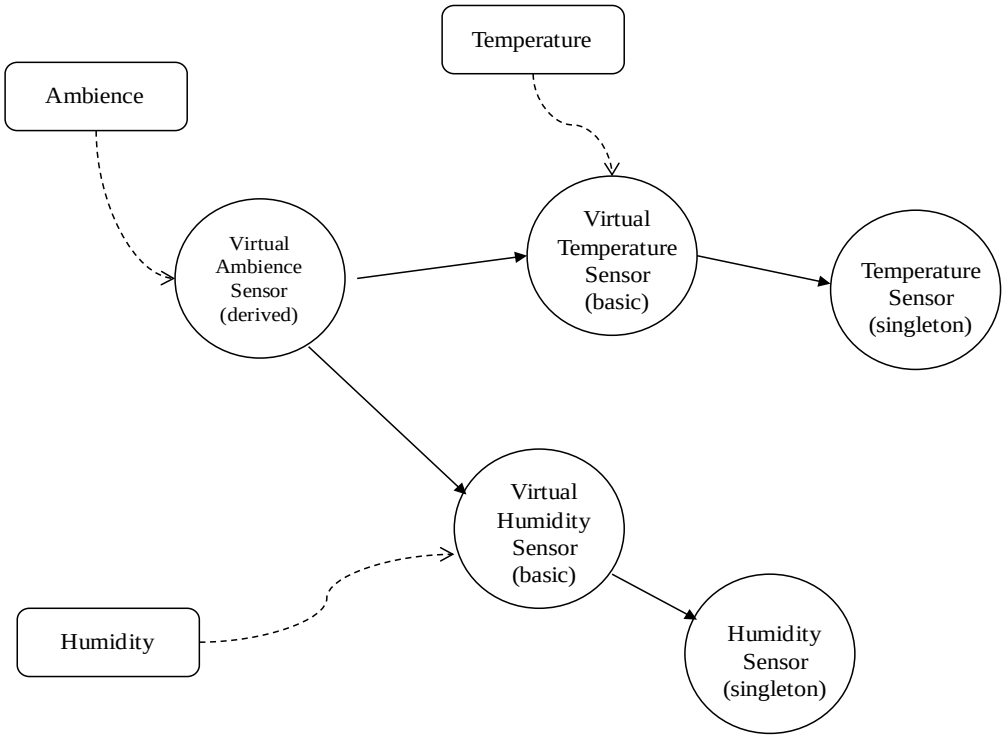

Figure 6-2. Sensor composition graph

Smart Space Ambience Sensor Example

We present an example scenario to demonstrate how all the components of the virtual sensor framework come together and work in unison to perform automatic on-the-fly service composition to enable the on-demand creation of virtual sensors.

Consider a sensor network consisting of temperature and humidity sensors deployed in a smart space which is divided into 4 rooms A, B, C and D (Figure 6-3). The knowledge base contains the sensor model definitions of the following virtual sensors: Singleton Temperature sensor (denoted by a solid black circle), Singleton Humidity sensor (denoted by a solid black square), Basic Temperature sensor (T), Basic Humidity sensor (H) and Derived Ambience (W). It also contains a user-defined data type definition <'Ambience', W> where W represents the model definition of the derived ambience sensor.

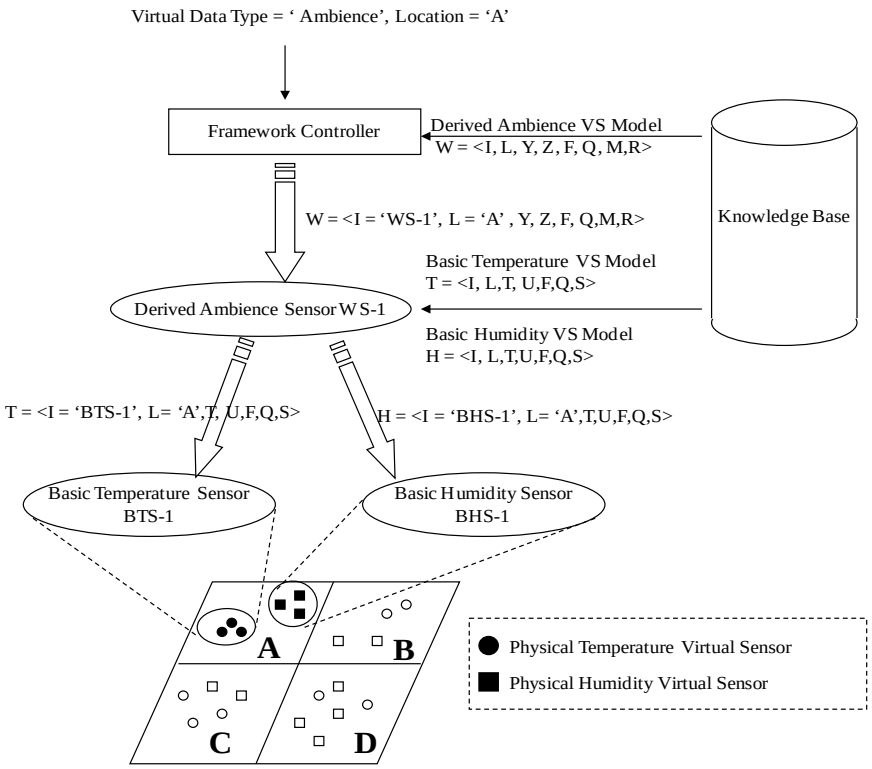

Figure 6-3. Sensing ambience using virtual sensors

Suppose a user sends the following query to the query processor: SELECT 'Ambience' FROM 'A'. The query processor extracts the virtual data type, 'Ambience' and the location, 'A' from the query, which it forwards to the framework controller. The framework controller then looks up the knowledge base to retrieve the phenomenon definition for 'Ambience'. Since the phenomenon name is hash-indexed to the virtual sensor which has the capability to sense

ambience, it retrieves the reference for the node representing the derived ambience sensor. From the node it gets the sensor model definition for the derived ambience sensor. It sets the location attribute in the model definition to 'A' and creates a sensor service object representing an ambience sensor. From the outbound edges of its node in the sensor composition graph, the ambience sensor knows that it needs to get data from basic temperature and humidity virtual sensors located at 'A'. The ambience sensor then retrieves the sensor model definitions of these basic virtual sensors and sets their location attributes to 'A'. It then uses the model definitions to create a new service object representing each basic virtual sensor. Each of these basic virtual sensors then figures out from the sensor composition graph that they need to aggregate readings from singleton virtual sensors of their respective types, which are located inside 'A'. Using mechanisms for service discovery and composition provided by the underlying service framework, the basic virtual sensors are able to access and aggregate data from the singleton temperature and humidity sensors. Once all the member virtual sensors are active and data starts flowing back up the chain, the ambience sensor becomes active.

Detecting the Enhanced Sentience of a Smart Space

The role of the sensor composition graph is not limited to the creation of virtual sensors. In fact, it plays a very important role in enabling a smart space to automatically detect whenever its sentience is enhanced due to introduction of new physical sensors into the space. The framework controller maintains two lists of virtual sensors: (1) Potential List: A list of sensors which can be created and; (2) Active List: A list of active sensors which have been created. The active sensors are created on-demand by traversing the sensor composition graph, as described previously. However, to maintain the potential list essentially requires a reverse-lookup of the sensor composition graph. When a new physical sensor connected to an Atlas node is introduced into the smart space and powered on, it automatically becomes available as a singleton sensor. The

virtual sensors framework then performs a reverse-lookup by traversing all the edges inbound into the node corresponding to the sensor type of the new sensor. This process is recursively done till all back edges have been traversed.

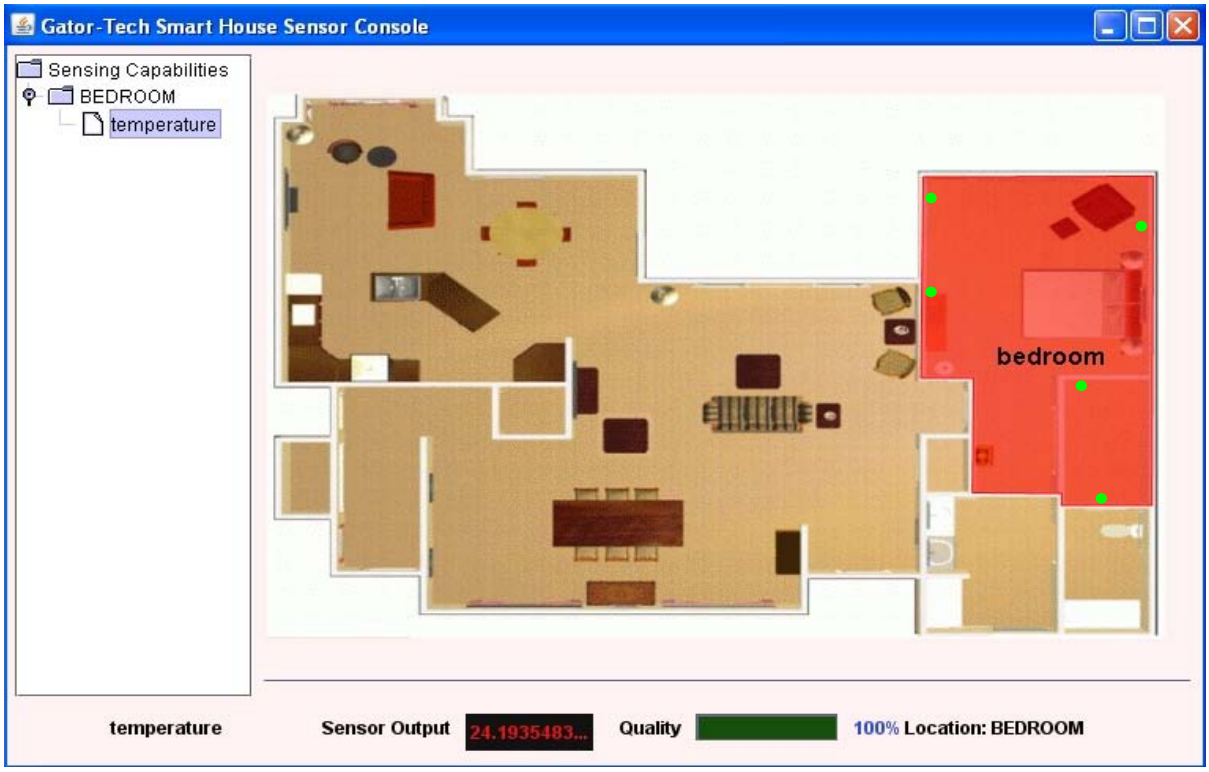

Figure 6-4. Initial sensing capability of the smart space

The framework controller adds any basic virtual sensors which is part of the traversal path into the potential list. Moreover, it also adds into the potential list, any derived virtual sensors which have all their member sensors in the active list or potential list. Consider the ambience sensor example described previously and assume that initially only temperature sensors are deployed in the space (Figure 6-4). In that case the smart space's initial sensing capability is limited only to temperature. However, as soon as humidity sensors are introduced into the space, the virtual sensors framework is able to detect that not only can the smart space detect humidity and temperature but can now also detect ambience (Figure 6-5). But one must note that, none of the virtual sensors are actually created till a user application explicitly issues a query involving

one of them. In this manner, it is able to prompt the user whenever the smart space's sentience gets enhanced both in terms of new primitive and virtual data types.

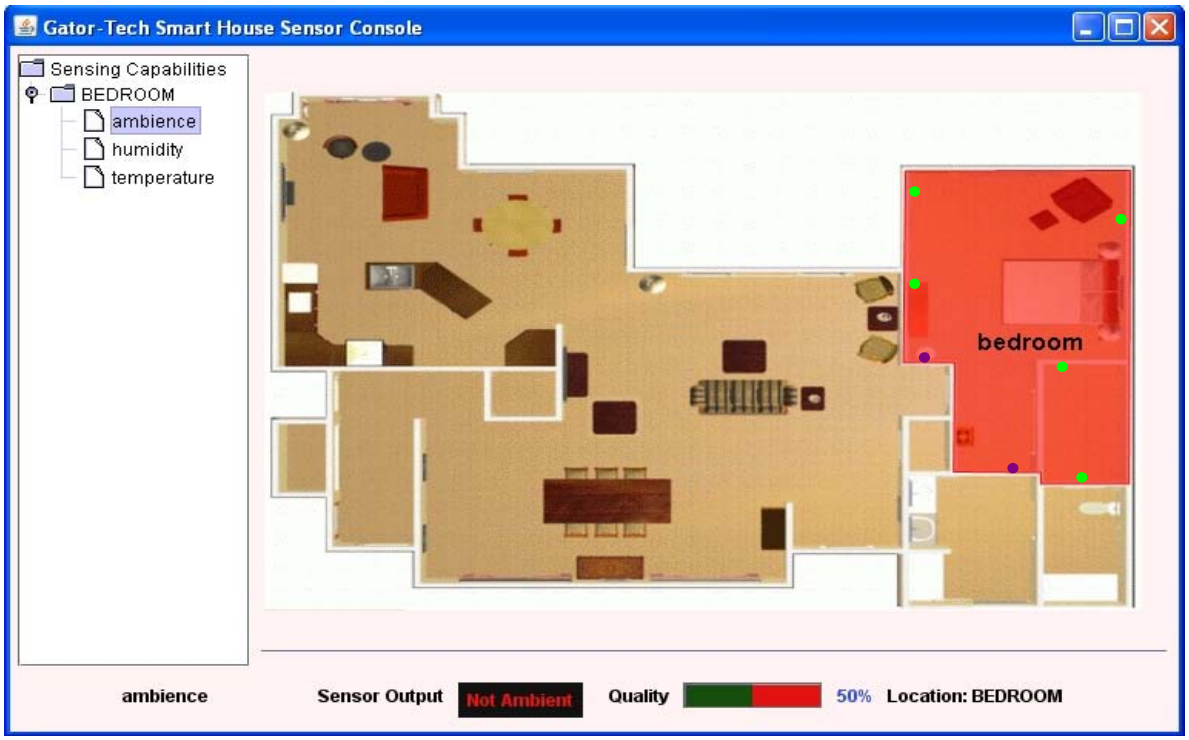

Figure 6-5. Effect of introducing humidity sensors into the smart space

Operations of a Basic Virtual Sensor

When the user queries for a basic virtual data type, the service layer looks up a knowledge base in the virtual sensor framework to find out which virtual sensor to create. This is done using sensor definitions stored in the knowledge base and using standard SOA mechanisms to compose the required singleton/physical sensor services. The service composition takes place in the centralized service layer and the naïve way would be to stream up all the data from each of the physical sensors for centralized aggregation. However this is clearly not scalable when the number of sensors involved is large. Furthermore, streaming up all the data through a multi-hop mesh network will lead to excessive latency and waste of network bandwidth and node energy.

To address the issue of scalability, we push down the process of basic virtual sensor aggregation from the service layer on to the sensor nodes so that it runs in a distributed in-

network fashion without requiring centralized processing. The aggregation is done using an aggregation tree overlaid over the topology of the network. Such aggregation techniques have been previously proposed in the context of ad-hoc networks [22] and our mechanism follows a similar philosophy since Atlas nodes support ZigBee-based mesh networking. The basic virtual sensor service object created in the service layer sends a tree-construction packet to the coordinator containing the identifier of the basic virtual sensor, its sampling rate in epochs and a list of its member sensor nodes represented by their network addresses.

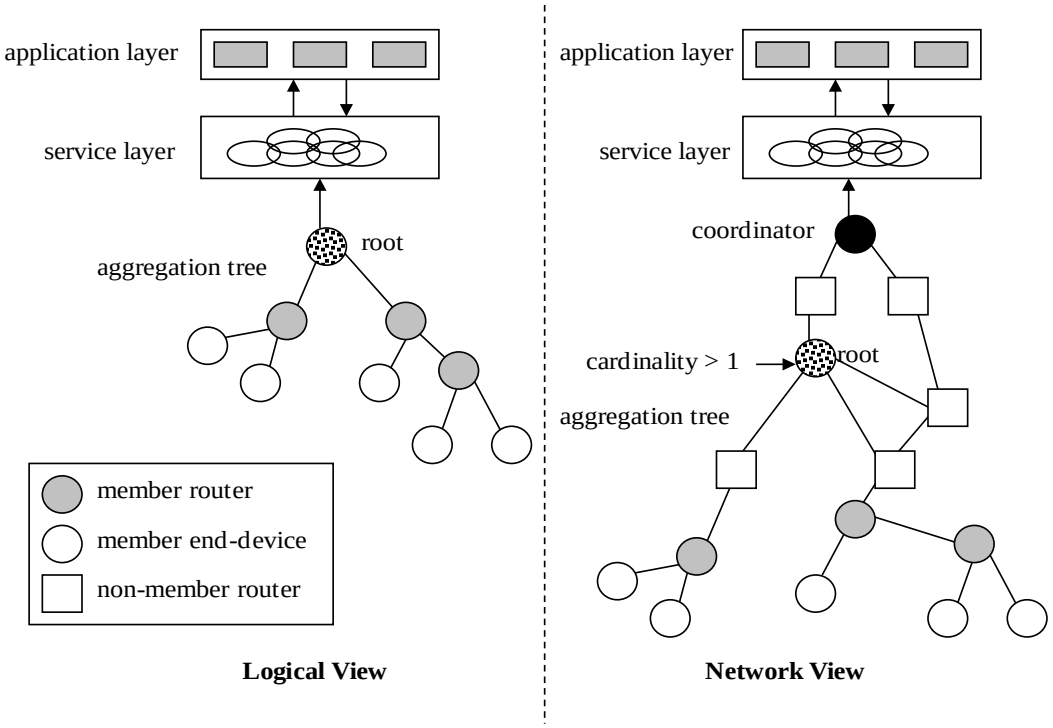

Figure 6-6. Logical versus network view of an aggregation tree

Before transmitting the packet, the coordinator applies its routing algorithm on the list of network addresses to determine the non-duplicate set of child nodes that it needs to propagate the message to. A hash function which uses network addresses as keys can be used for this purpose. This ensures that each recipient node does not receive duplicate copies of the tree-construction packet as a consequence of multiple member nodes sharing the same routing path. Each router node in the network which receives the packet continues this process. Each router is also able to

determine if it is the root of the virtual sensor aggregation tree or not by looking at the set of child nodes that it needs to propagate the packet to. The first router whose set of recipient child nodes has cardinality greater than 1 (Figure 6-6, Network View), becomes the root of the tree. This follows from the fact that the message only travelled along a single routing path before reaching this node and now for the first time the route gets split into multiple sub-paths. In order to indicate to other nodes that a root has been selected, this node sets a special bit in the message packet before forwarding it further down the network. In case the coordinator is the root node it sets this special bit before forwarding the initial packet. Whenever a router or end-device finds its address listed in a tree-construction packet, it removes its address from the list and also extracts the basic virtual sensor id along with its associated sampling rate. In this manner the aggregation tree is constructed on top of the ZigBee network. However, one must note that the logical view of the tree may be quite different from its network view as we can see in the example (Figure 6-6, Logical View). The network view of the tree might contain router nodes whose sensors are not members of the basic virtual sensor and whose only function is to act as a link with different segments of the aggregation tree. This is a consequence of self-organizing networks, where nodes connect to their nearest routers based on proximity and signal strength.

In case of proprietary ad-hoc routing protocols, it might be possible to tweak the algorithms to force nodes to form a network in a certain way. Semantic Routing Trees (SRTs) [34] used in conjunction with link-based parent selection by TinyDB, provide such a functionality. However, this implies that the network has to be re-organized whenever the index attributes change. This is not only expensive but also of limited utility when multiple applications are running simultaneously. Moreover, such techniques require modification of networking layers in an application-specific manner which may not be advisable.

Basic Virtual Sensor Aggregation Process

The aggregation mechanism consists of propagation of partial records starting from the leaf nodes all the way to the root of the tree. Every time a participating end-device samples its sensor, it creates and sends a partial aggregation record to its parent. Each router on receiving a partial aggregation record from its child nodes merges them, updates it if necessary and forwards it to its parent. For example, if the aggregation formula being used was the arithmetic mean then a node would send a partial aggregation record <Sum, Count>, where Sum denotes the partial sum of readings collected and Count denotes the number of sensors sampled for the sub-tree rooted at this node. The root node on receiving the aggregation record calculates the final output, for example, for arithmetic mean it calculates the output as Sum divided by Count.

Fault Tolerance of Basic Virtual Sensors

Failure is an integral part of any sensor network especially for large scale deployments. Whenever a member node (and the sensor physically connected to it) fails, the basic virtual sensor loses a data source and this affects its data quality.

A basic virtual sensor consisting of N Singleton Virtual Sensors, outputs a single reading, which is an aggregate of the N readings obtained from them. In case of failure of one or more singleton sensors, it is likely that the basic virtual sensor can still continue to function albeit at the cost of providing data of a lower quality. We provide an approximation algorithm which enables basic virtual sensors to be fault tolerant up to a certain number of sensor failures, depending on user preferences. For derivation purposes, we assume that the aggregation function of the basic virtual sensor is the arithmetic mean. We begin by defining the following terms:

- **N** = Number of singleton virtual sensors which are initially members of the basic virtual sensor.
- **S** = $\{s_i\}_{i = 1 \text{ to } N}$: Set of the singleton virtual sensors.

- $\mathbf{S_F}$ = Set of dead singleton virtual sensors.
- $\mathbf{N_C}$ = Number of singleton virtual sensors currently alive.
- $\mathbf{DQ_T}$ = Minimum Data Quality Threshold.
- $\mathbf{T_{start}}$ = Time when the sensor network was started up.
- $\mathbf{R_{s_i}}$ = Reading from sensor $s_i \in S$.
- $\mathbf{N_t}$ = Number of readings sampled till time t.
- **B** = N x N matrix where B_{ij} = number of instances when $|R_{s_i} - R_{s_j}| < \varepsilon$ (ε is dependent on the sensitivity of the sensor hardware and is typically chosen between 5 and 20). B_{ij} gives the number of observed instances when sensors s_i and s_j exhibited similar behavior. B is updated whenever the sensors are sampled.
- $\mathbf{P_{s_i\text{-}s_j}}$ is the probability that sensors s_i and s_j behaved in a similar manner and depends on the time t at which it is calculated. $P_{s_i\text{-}s_j} = \frac{B_{ij}}{N_t}$.
- $\mathbf{W_{s_i}}$ = Probabilistic weight associated with R_{s_i} where,

  $\mathbf{W_{s_i}} = \begin{cases} 1 & \text{; if } s_i \text{ is alive.} \\ W_{s_j} P_{s_i\text{-}s_j} & \text{; if } s_i \text{ is dead and its readings are approximated using } R_{s_j} \text{ for some } j \neq i. \end{cases}$
- $\mathbf{A_{s_i}}$ = Set of sensors whose readings are being approximated using readings from sensor s_i. Initially for all i = 1 to N, A_{s_i} is set equal to $\varnothing$.
- $\mathbf{VS_{reading}}$ denotes a reading from the basic virtual sensor and is calculated by Equation 6-4.

$$VS_{reading} = \frac{\sum_{s_i \in S} W_{s_i} R_{s_j}}{\sum_{s_i \in S} W_{s_i}} \qquad (6\text{-}4)$$

We also make the following assumption. All singleton virtual sensors which are members of a particular Basic Virtual Sensor represent physical sensors having the same characteristics such as sensitivity, error and range of output values. All singleton virtual sensors are sampled together periodically in a synchronized fashion. All singleton virtual sensors are alive at startup

and hence, initially $W_{s_i} = 1$ for i = 1 to N. Suppose S' is the set of singleton virtual sensors that fail at time t then the approximation algorithm is given as follows:

1. Set $N_C = N_C - |S'|$.

2. For each sensor $s_x \in S'$, compute the following:
   i) Calculate P_{s_x} equal to max $\{\{0\} \cup \{P_{s_x\text{-}s_j}$ for all $s_j \in S - S' - A_{s_x} \mid P_{s_x\text{-}s_j} > P_T\}\}$ where P_T is the minimum user-defined threshold probability. P_T ensures that the maximum observed probability of similar behavior is high enough not to be attributed to chance. Note that in case none of the probabilities exceed P_T, then $P_{s_x} = 0$. Suppose P_{s_x} is equal to $P_{s_x\text{-}s_y}$ for some y. This implies that sensor $s_y \in S$ behaved most similar to s_x.

   ii) Set W_{s_x} equal to $W_{s_y} P_{s_x}$ and A_{s_y} equal to $A_{s_y} \cup \{s_x\}$. Hence, from now onwards, readings for sensor s_x will be approximated using readings from sensor s_y, which implies that R_{s_x} is equal to R_{s_y}. We assume that the equations for calculating W_{s_x} and R_{s_x} are stored in the framework. This ensures that: (a) Subsequent to each sampling, sensor readings for s_x are generated using data obtained from sensor s_y and; (b) Whenever W_{s_y} changes (for example, when sensor s_y fails), W_{s_x} also gets updated.

3. Update W_s for all $s \in A_{s_x}$ and set S_F equal to $S_F \cup \{s_x\}$.

4. If $W_{s'}$ less than P_T, for any $s' \in S$, set $W_{s'}$ equal to 0 and set S equal to $S - \{s'\}$.

This approximation algorithm ensures that during computation of $VS_{reading}$ the absence of sensors is compensated by readings obtained from other live sensors thereby increasing availability of sensor data while reducing the loss of data quality.

Each router in the aggregation tree runs the fault tolerance algorithm described above and maintains a history of similar behavior among its children who are members of the basic virtual sensor. For example, if a router has 3 member sensors i, j and k as its children then it maintains three variables s(i, j), s(i,k) and s(j,k). If we use Euclidean distance as a measure of similar behavior then, during any sampling epoch, whenever the absolute difference between $reading_i$ and $reading_j$ is less than ε (where ε is the sensitivity of a sensor), s(i, j) is incremented by 1. Similarly s(i, k) and s(j, k) are also updated as required. The larger the value of these similarity statistics, the better is the quality of sensor-based approximation. If a child sensor node fails, its

parent router automatically approximates its readings by choosing a live child node which was observed to be behaving most similar to the failed sensor. This choice is made based on the statistics collected till that point of time. Suppose in our example sensor i fails then another live sensor (call it x) is used to approximate its readings, where x is equal to max_y s(i, y) with y is equal to {j, k}. The variables used for maintaining similarity statistics are periodically flushed to ensure that they do not get skewed by outdated information. This sensor based approximation mechanism not only has low processing overhead as compared to model-based approximation techniques but also compensates for the loss of sensors without requiring assistance from the service layer. Approximating readings of failed sensors may result in an improvement in the quality of virtual data as compared to having no compensation at all; however, it will still result in a drop in the overall data quality. Failure among member nodes of a basic virtual sensor can be one of three types:

1. End-device failure: The router which is the parent of the end-device detects this situation as a consequence of the ZigBee sync mechanism and executes the approximation algorithm described above.

2. Router failure: The parent of the router executes the approximation algorithm while its children on detecting the disconnection, try to re-connect to an alternate router available in the network utilizing the self-healing features of ZigBee. On re-establishing connection, a node sends a message to the new router announcing its basic virtual sensor id and aggregation formula for merging partial records. This router then adds them as child nodes and takes on the responsibility of collecting similarity statistics and executing the fault tolerance algorithm if any of them fails.

3. Root of aggregation tree fails: If the root node of the aggregation tree fails, then a new root needs to be discovered. We propose a root discovery algorithm which guarantees that at most one root gets selected even if several nodes are in contention for that role. Each of the failed root's children initiates the discovery process by first transmitting a packet to its other siblings to determine if they are alive or not. After waiting for a certain period of time, it transmits a root-search packet containing its virtual sensor id and number of live siblings. If required, it also contains the sensor ids and network addresses of root nodes of the derived virtual sensors which get inputs from this basic virtual sensor. This packet has its destination address is set equal to 0 which is the network address assigned to a coordinator by default. Each router forwards this message to its parent and caches it. However, if any router

receives messages from all the siblings it considers itself as the new root of the aggregation tree. It suppresses further transmission of the root-search packet and sends a new-root message to each sibling declaring itself as the new root node. Since it is possible that due to time delays and synchronization errors, multiple nodes can declare themselves as the root of the same tree, each router which considers itself the root node also suppresses forwarding of any new-root messages. This ensures that only the node with the shortest network path to all the siblings becomes the new root. In case the basic virtual sensor is a member of a derived virtual sensor, the new root node is also responsible for linking up with the root of the derived virtual sensor.

Monitoring Data Quality of Basic Virtual Sensors

The fault tolerance mechanism approximates readings of failed sensors by using readings from other live sensors. This naturally affects the quality of data being output by the virtual sensor. In order to monitor data quality, each reading originating from a basic virtual sensor is associated with a virtual sensor quality indicator ($VSQI_{BVS}$) value. This value provides a measure of the virtual sensor's data quality. $VSQI_{BVS}$ is a function of the number of live sensors contributing readings to the basic virtual sensor and the number of failed sensors whose readings are being approximated using live sensors by the fault tolerance mechanisms. The formula for computing $VSQI_{BVS}$ is given by Equation 6-5.

$$VSQI_{BVS} = \frac{(N_C + \sum_{s \in S_F} g(\Delta t_s) W_s)}{N} \quad \text{(6-5)}$$

where, $g(x) = 1 - e^{-x/a}$ and $VSQI_{BVS} \in [0, 1]$

$VSQI_{BVS}$ has a maximum value of 1, if all member sensors are alive, and 0 if all the sensors are dead. N is the number of live physical sensors which were originally members of the basic virtual sensor. N_C denotes the number of physical sensors which are currently alive. S_F is the set of failed sensors. Δt_s is the time elapsed between the start up and failure of a failed sensor 's'. W_s is a weight associated with sensor 's' where, $W_s \in [0, 1)$. W_s is computed by the fault tolerance mechanism, as the probability that the live sensor being used for approximation behaved similar to the failed sensor. The term g(x) is a time dependent weight function

associated with W_s, where '*a*' is a constant greater than 0. The value of '*a*' determines how fast g(x) converges to 1 and depends on the sensors and their deployment environment. If W_s is calculated based on observations taken over a long time interval, then that value of W_s will be given more weight than say, a value of W_s which has been calculated based on observations taken over a shorter period of time.

The value of $VSQI_{BVS}$ is also computed using the partial aggregation method described before. A partial aggregation record for $VSQI_{BVS}$ is given by $<Q_P>$. Q_P denotes VSQI information, for the sub-tree (call it S_T) rooted at this node. The Q_P term is given by $N_L + \sum_{s \in S_F \cap S_T} g(\Delta t_s) W_s$, where N_L denotes the number of live sensors in S_T and g(t), t_s and W_s are as defined above. The final $VSQI_{BVS}$ value is calculated at the root as Q_P/N, where N is as defined above. For practical purposes, the partial aggregation records for sensor readings and $VSQI_{BVS}$ are not transmitted separately but instead are merged into a single record: <Sum, Count, Q_P>.

Operations of a Derived Virtual Sensor

The role of a derived virtual sensor is to apply a fusion function on inputs from multiple heterogeneous virtual sensors (whether basic or derived) to produce more sophisticated and abstract types of data. This enables it to answer queries on data types which cannot be processed by the other virtual and physical sensors such as ambience and complex events like activities of daily living.

Intuitively a derived virtual sensor should be residing solely inside the service layer since it is not directly bound to inputs originating from any physical sensor. However, this will lead to scalability and latency issues due to the centralized service layer. In order to address these issues, we push down the operations of a derived virtual sensor on to the distributed mesh network of nodes. When a user queries for a derived data type, the service layer looks up the definition of

the associated virtual sensor from the knowledge base. It then composes together the different member virtual sensor services into a derived sensor service and initiates the process of creating an in-network aggregation tree. First the derived virtual sensor service obtains the network addresses of the root nodes of each of its member sensors. This is possible due to the fact that the packets containing the output of any virtual sensor will have its source address equal to that of the root node of that sensor's aggregation tree. Then a sensor-construction packet containing the list of these network addresses is dispatched to the ZigBee network via the coordinator. In a manner similar to the one described previously for basic virtual sensors, each router applies its routing algorithm on the list of network addresses to determine the non-duplicate set of child nodes that it needs to propagate the message to. The first router who's set of recipient child nodes has cardinality greater than 1, becomes the root of the aggregation tree for the derived virtual sensor. This node becomes responsible for applying sensor fusion to the data originating from sensors which make up the derived virtual sensor.

Derived Virtual Sensor Aggregation

The aggregation of readings in a derived virtual sensor is pretty straightforward, with each member sensor sending its output to the root node which computes the final output and transmits it to the service layer via the coordinator. Unlike a basic virtual sensor which aggregates data from homogeneous sources, a derived virtual sensor has to operate on multiple heterogeneous inputs. Hence, the partial aggregation techniques used for basic virtual sensors are not utilized for derived virtual sensors.

Fault Tolerance of Derived Virtual Sensors

By virtue of its definition, a derived virtual sensor cannot be too resilient when it comes to tolerating failures. We take the position that if a member sensor fails completely then the derived virtual sensor should cease to operate due to lack of one of its inputs. Since we are not focusing

on a particular application scenario for derived virtual sensors, we are forced to adopt such a pragmatic stance. It is entirely possible that in some application domains derived virtual sensors may be able to operate without the presence of one or more inputs. However, in the interests of providing a generic framework for virtual sensors, we decided to adopt an 'all-or-none' approach in the context of derived virtual sensors. We hope to rectify this as part of future work. We discussed most of the failures when we covered fault tolerance for basic virtual sensors. However, if the root of a derived virtual sensor's evaluation tree fails we need to take special measures. In this case, the root of each member virtual sensor sends a root-search message addressed to the coordinator. Any router receiving messages from all the root nodes of the members considers itself the new root of the derived virtual sensor and sends a new-root message to the sender nodes. It also suppresses further transmission of root-search and new-root messages.

$$VSQI_{DVS} = \prod_{1 \le i \le n} VSQI_{BVS_i} \times \prod_{1 \le j \le m} VSQI_{DVS_j};\ VSQI_{DVS} \in [0, 1] \qquad (6\text{-}6)$$

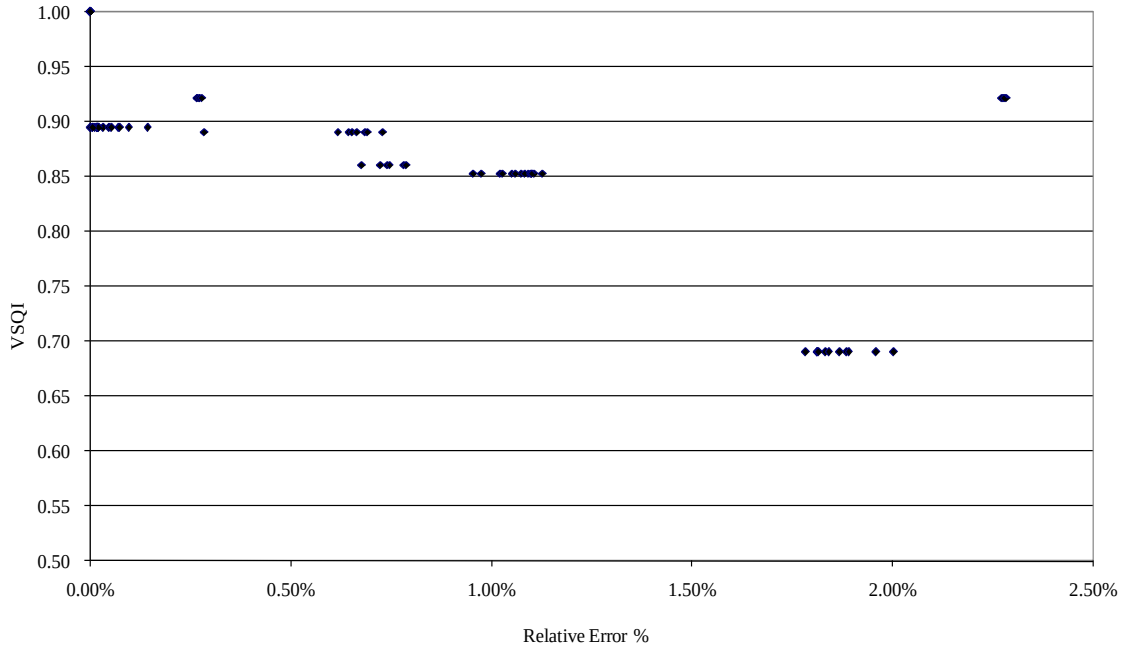

Figure 6-7. Relative error % vs. VSQI

Monitoring Data Quality of Derived Virtual Sensors

A derived virtual sensor may be composed of multiple heterogeneous basic or derived virtual sensors; hence, monitoring the performance of a derived virtual sensor requires a slightly different approach than that of a basic virtual sensor. The $VSQI_{BVS}$ formula gives the probability that the approximated output of a basic virtual sensor matches its expected output, if it was fully functional. Following the same trend, we defined VSQI associated with a derived virtual sensor as the product of VSQIs of its member basic and derived virtual sensors (Equation 6-6).

Equation 6-6 reflects the fact that in case of a derived virtual sensor there is no concept of compensating for the loss of one of its member sensors. If one of the member basic or derived virtual sensors fails then its parent derived virtual sensors also fail.

Experimental Performance Analysis

We analyze the performance of the virtual sensor mechanisms presented in this paper, using both simulated and real-world data. We conducted two sets of experiments, one for evaluating fault tolerance and data quality monitoring and the other for analyzing the performance of our methods as compared to centralized techniques, in terms of latency and energy consumption of the sensor nodes.

Fault Tolerance and Data Quality Monitoring

The goal of the first set of experiments was to measure the effectiveness of the fault tolerance algorithm and data quality monitoring. We were interested in evaluating whether the sensor-based approximation method was an effective way of compensating for failed sensors and if the virtual sensor quality indicator (VSQI) accurately reflected the effect of the approximation scheme on overall data quality.

We used a real-world data set from Intel Research Lab, Berkeley for conducting this set of experiments. This data set consists of temperature, humidity and light level readings collected

from 54 nodes deployed at the Intel Research Lab in Berkeley, California, over a period of one month. Out of the 54 sets of data, only 40 were short-listed as the remaining ones had incomplete sets of readings. Each sensor log was also sorted to ensure that all the readings were ascending order of their time stamps.

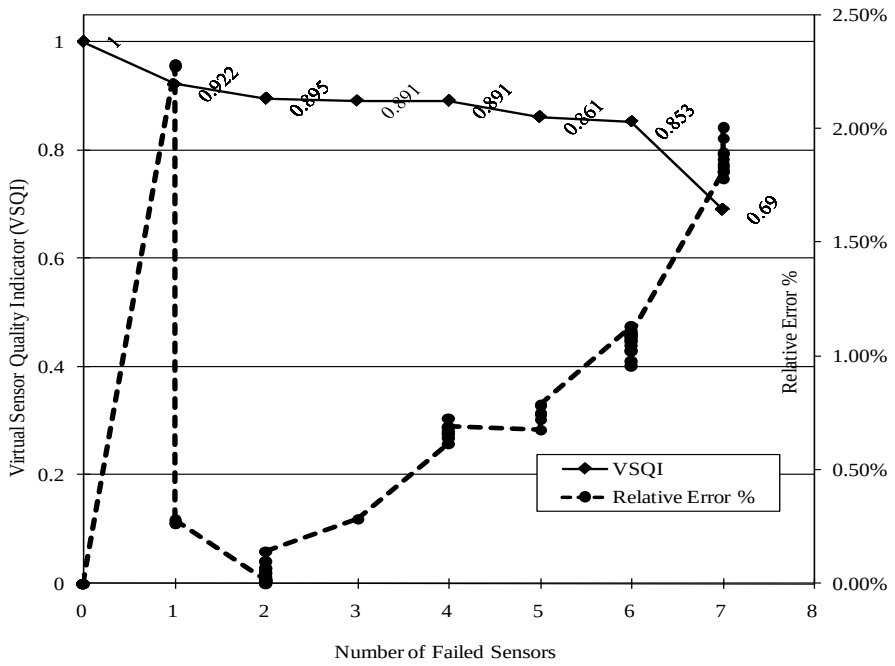

Figure 6-8. Number of sensor failures vs. VSQI and relative error

Based on sensor location information provided in the documentation, we created basic virtual sensors composed of 8 physical temperature sensors which used arithmetic mean for aggregation. Sensor failures are simulated at periodic intervals and the VSQI value and percentage relative error logged for each epoch of sampling. The relative error value was computed as the deviation of the actual output of the virtual sensor (based on aggregation of live and approximated sensor readings) from the expected output (based on aggregation of readings from all live sensors). Since at the time of logging of the original data sets, all sensors were functioning properly, the induced failure allows us to evaluate how well the virtual sensors compensate for it.

To determine the effectiveness of data quality monitoring we explored the correlation between VSQI and the relative error. As described earlier, VSQI is a measure of the quality of a virtual sensor; in terms of the probability that compensated values accurately reflect the readings should all sensors remain functional. Relative error is a different measurement of the quality of virtual sensor, which focuses on the deviation of the output from the expected value because of the internal compensation within a virtual sensor. Relative error can be heavily dependent on the specific datasets and the configuration of the virtual sensors. However, basic virtual sensors components deployed in realistic settings are usually spatially correlated, which leads to smaller and bounded relative errors. We observe that in general, the VSQI and the relative error are inversely proportional to each other (Figure 6-7). As the relative error increases (usually due to more failures and heavier compensation), the VSQI decreases accordingly. From the plot, one also observes that when the relative error is approximately 2.3%, its corresponding VSQI value seems to be quite high. Upon inspection of the simulation logs, it was determined that this anomaly occurred due to temporary malfunctioning of one of the compensating sensors, which resulted in the output of spurious readings for a short duration.

Next, we studied the effect of sensor failure on VSQI (Figure 6-8). The simulation showed that as more singleton virtual sensors fail the VSQI decreases. However, VSQI only decreases gently as sensors fail, which demonstrates the fault-resiliency of the virtual sensor mechanism. Depending on the specific functioning sensor chosen to compensate for a failed one, the relative error may spike irregularly, but in general, the relative error increases as more sensors fail. One can also observe that, there is a temporary increase in the relative error when the first sensor fails. Upon further inspection of the logs, it was determined that the compensating sensor experienced some temporary malfunction and as a result output spurious readings for a short

duration, before resuming normal operation. As mentioned earlier, even under extreme wide-spread failures (87.5%), the relative error always remains bounded (less than 2.5%), which demonstrates the effectiveness of fault tolerance in virtual sensors.

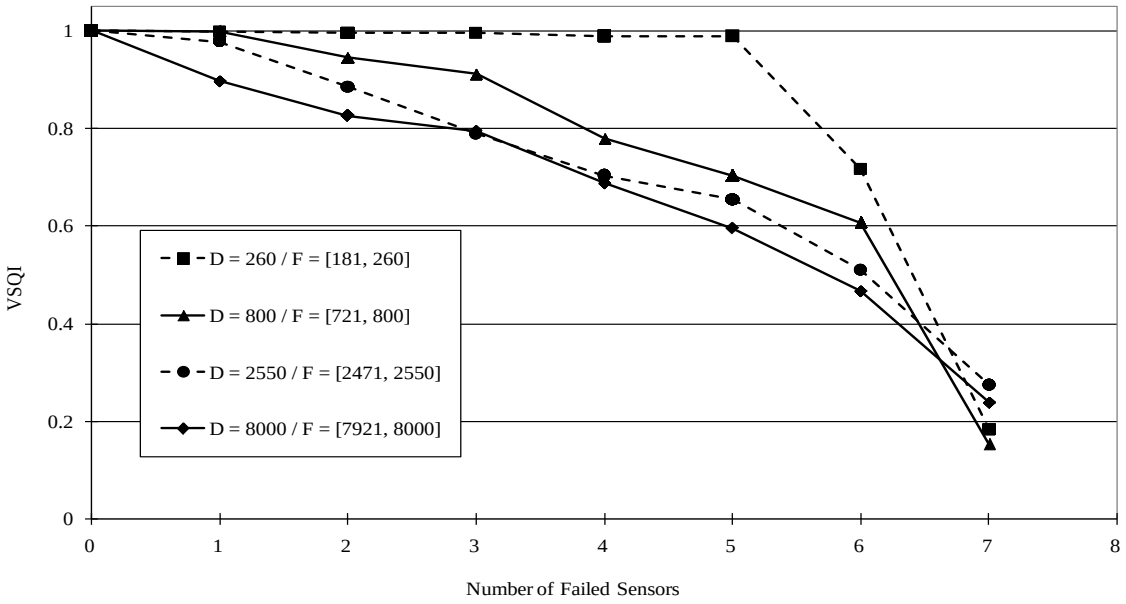

Figure 6-9. Effect of sensor failure pattern on VSQI

The results of the simulation also show the effect of failure patterns on VSQI (Figure 6-9). We simulated random singleton virtual sensor failures beginning at time 0, 180, 720, 2470 and 7920, with the stipulation that all the sensors will fail within 80 epochs. The purpose was to explore whether allowing the sensors to accumulate a long correlation history improves VSQI when sensors start to fail. When the choice of the converging time constant 'a' is kept low (we chose a=4), the weight of approximation plays a more dominating role. We observed that the later the sensor failures are introduced, the sharper the drop in VSQI when sensors do fail. This phenomenon appears to be contradictory to our intuition at first, but upon further analysis, it is found that since the sensors that are used to compensate are not always placed at the exact same locality, the probability that two sensors exhibiting comparatively similar behavior always generate readings of equivalent magnitude, decreases as the history accumulates over time.

Latency of Data Arrival and Energy Consumption

The second set of experiments was conducted to evaluate system performance in terms of latency in arrival of virtual sensor output and energy consumption of nodes per epoch of sampling. We compared the performance of our distributed virtual sensor mechanisms versus the mechanism where virtual sensors are executed exclusively inside the service layer of the SOSN and perform centralized aggregation on data streams. The Global Sensor Network (GSN) [1] is example of a virtual sensor framework which utilizes this form of centralized streaming approach for implementing virtual sensors.

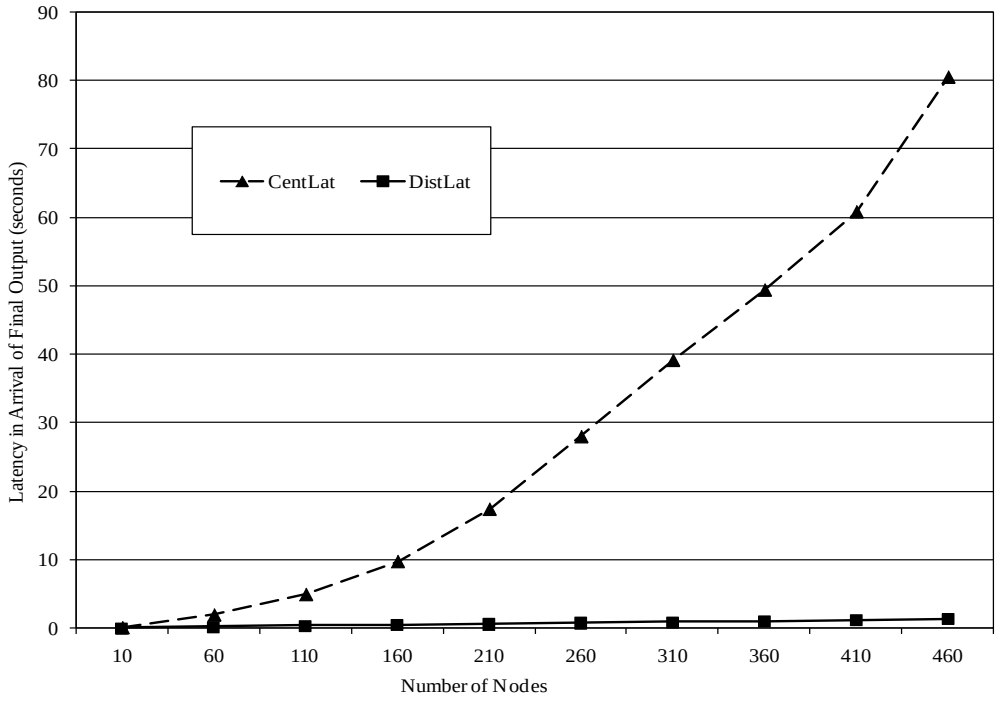

Figure 6-10. Comparing latency in arrival of final output

For distributed tree-based aggregation, we used a simulator to randomly generate aggregation trees overlaid on top of a mesh network. The simulator takes as input the number of sensor nodes which are members of a virtual sensor and randomly generates an aggregation tree of nodes for each iteration. To simulate the overlaying of the tree on top of a mesh network, it randomly assigns to each node one of the following roles: (1) end-device which is a member of

the virtual sensor; (2) router which is a member of the virtual sensor; and (3) router which is not a member of the virtual sensor (that is, it only links different segments of the aggregation tree).

To calculate latency of virtual sensor output, we assume a node takes 0.003 seconds to receive or transmit a packet (based on a packet size of 100 bytes and ZigBee transmission bit rate of 250 Kbps). For computing energy consumption of nodes we obtained power consumption data from Atmel for their Zlink RCB nodes. We use a power consumption cost model where each member end-device and router's cost is calculated as the sum of processing, sensor sampling and transmission costs. Additionally each member router also incurs the cost of receiving messages. On the other hand, each non-member router's cost is simply the cost of receiving and transmitting messages. We observe that in case of centralized aggregation (denoted by 'CentLat'), the latency increases significantly as the number of sensors increases (Figure 6-10). This is due to the fact the each sensor reading has to traverse multiple hops all the way up the tree in order to reach the service layer and the aggregation operation cannot take place till readings from all the member sensors have streamed up. One would assume that since messages flow up the tree in parallel, latency of the streaming mechanism will be low. However, this assumption turns out to be invalid in a sensor network which uses multi-hop networking. This is due to the fact that in the naïve streaming approach, each sensor reading packet travels all the way up to the base station and there is no merging of packets at any routing node. Hence, at each hop a packet has to wait till the radio of that router node is free to transmit before it can propagate up the tree. This wait time can be long if there are a number of child nodes which are waiting to transmit their packet via this router node since the radio of the node can only transmit one packet at a time. Therefore, with each hop the latency of the packet increases and since every sensor reading packet has to propagate all the way up the tree, an increase in the number of

sensors leads to a drastic increase in latency of response for the virtual sensor. The distributed aggregation tree mechanism (denoted by 'DistLat') on the other hand exhibits extremely low latencies of approximately 1 second or less, even when the number of sensors is large. This can be explained by the fact that a reading from each sensor node only travels only 1 hop to its parent aggregation node where it is merged with other partial aggregation records. Hence, readings from each sensor do not need to be individually transmitted over multiple hops all the way to the service layer thereby, significantly reducing latency of virtual sensor output. Moreover, due to its lower utilization of network resources, it reduces the processing and networking costs in intermediate routers, which translates into significant reduction in overall energy consumption.

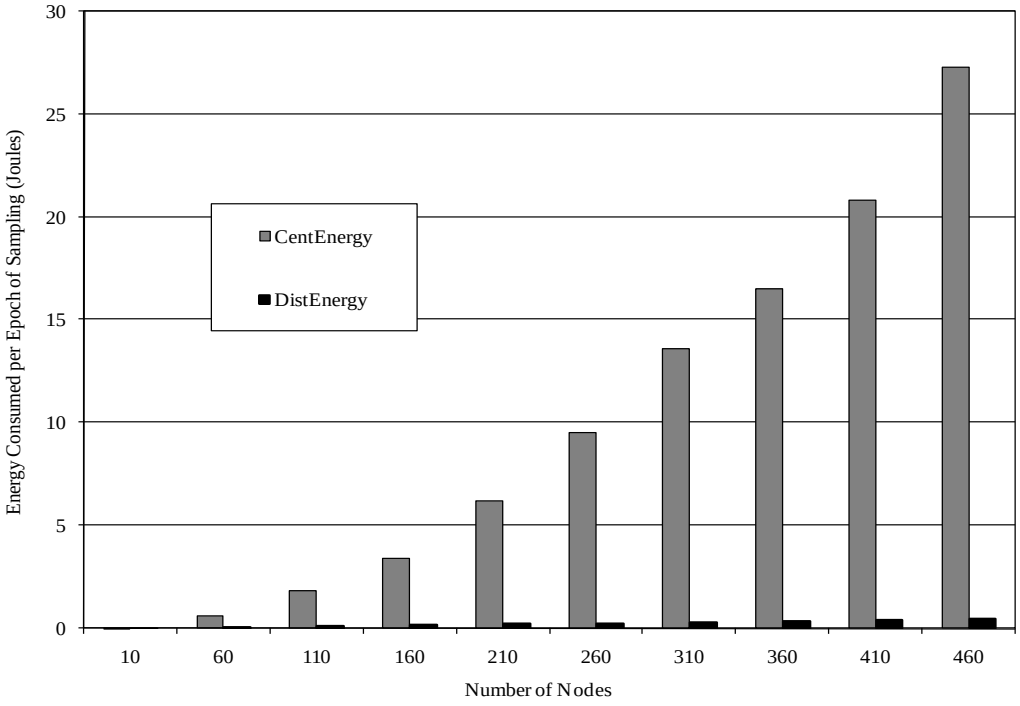

Figure 6-11. Energy consumption of virtual sensor per epoch

With regards to total energy consumption, we observe that for the centralized method (denoted by 'CentEnergy'), the total energy consumed by nodes for each epoch of sampling dramatically increases as the number of member sensors increases whereas for the distributed mechanism (denoted by 'DistEnergy') there is no significant increase in energy consumption

(Figure 6-11). In fact the total energy requirements per epoch for the distributed mechanism is anywhere from 90-98% less than the centralized method. This implies that the distributed mechanism is not only more scalable but is also more energy efficient as compared to the centralized method.

CHAPTER 7
PHENOMENA DETECTION AND TRACKING

Contemporary wireless sensor network (WSN) research done in the area of detection and tracking has primarily concentrated on and proposed distributed algorithms for, observing motion of objects whose shape and size are invariant [15, 25]. In contrast, phenomena clouds such as oil spills and gas clouds not only exhibit motion but are also characterized by non-deterministic variation of their shape and size, over time. However, the utility of phenomena cloud detection and tracking is not restricted only to application domains involving gas clouds or oil spills. In fact, they can also be utilized in situations where the quality of data originating from individual sensors cannot be trusted in isolation. In such cases, the raw sensor data originating from the system is typically extremely noisy which makes it very difficult to distinguish actual events from random stimuli. Hence, a "quorum" of multiple sensors which are located in close proximity to each other is required to reduce the probability of false positives. Through our collective research and systems experience over the years in building Smart Spaces at University of Florida's Mobile and Pervasive Computing Laboratory, we have found great utility in applying the phenomena cloud concept for efficiently and accurately monitoring various events in the space.

Stream-based mechanisms such as Nile-PDT [2, 3], which have been proposed for detection and tracking of phenomena clouds do not take into account the cost of acquiring and transmitting sensor readings, and typically require participation from all sensors in the network. Unfortunately, sensor sampling costs and networking and processing overheads can have a critical effect on the practical viability of the entire smart space and hence there is a need for distributed in-network mechanisms, where execution of the detection and tracking process is localized to the immediate neighborhood of a phenomenon at any given time. Furthermore, due

to primitive processing capabilities of wireless sensor nodes such mechanisms should avoid relying on construction of mathematically complex cloud models. Hence, there is a need for localized, in-network algorithms which provide energy-efficient, real-time phenomena detection and tracking capabilities without requiring complex cloud models.

The remainder of this chapter is organized as follows. First we define phenomena clouds and their characteristics. We analyze their structure when overlaid on a sensor space to generate a set of parameters to comprehensively describe them without requiring a formal mathematical model. Next, we describe the critical challenges faced during the detection and tracking process. Then we describe energy-efficient, localized, in-network algorithms for real-time detection and tracking of phenomena clouds, which do not require customization of the network routing layers. The proposed algorithms work in an autonomous manner without requiring intervention from the centralized query processor residing in the base station and hence, are suitable for disconnected mode of operation, when continuous communication with the base station cannot be maintained. Next, we describe an interesting and practical real-world application of our phenomena detection and tracking algorithm in a smart space. The application described in this chapter is presently running in the Gator Tech Smart House, a real-world smart space, to solve critical challenges in detecting certain activities. Finally we evaluate and analyze the performance of our approach through real-life experiments and simulations. We also provide a comparison of the resource usage and energy consumption characteristics of our approach against contemporary real-world stream-based detection and tracking techniques such as Nile-PDT.

Phenomena Clouds

A phenomenon cloud can be defined as a manifestation of a number of simultaneous events reaching “critical mass” and spanning a contiguous space. The shape, size and direction of movement of such phenomena clouds either cannot be modeled accurately or have models which

are usually too complex for real-time computing by sensor networks, which largely consist of low-end nodes with limited processing capabilities. Examples of such phenomena include gas clouds, flooding, oil spills, fires or even movement of tourists in a museum. A phenomenon cloud can exhibit non-deterministic behavior over time making it difficult to anticipate its path and motion.

Major Challenges

Major challenges faced during detection and tracking of phenomena clouds are as follows:

1. Initial detection of phenomenon. Initial occurrences of the phenomenon might be scattered throughout the space. Hence, detection needs to be attempted at multiple locations.
2. Avoiding false positives. The probability of a single sensor outputting inaccurate readings at a specific point in time is not low. Hence, it is quite possible for a sensor to temporarily malfunction or be subject to environmental conditions which might cause it to output values which incorrectly indicate the occurrence of a phenomenon.
3. Tracking a phenomenon in real-time. A phenomenon can suddenly grow or shrink in size and also move in multiple directions simultaneously; hence, tracking it in real-time can become a massively complex task. To enable cost-effective real-time tracking, the rate of status updates from the sensor network to the user needs to be kept at a minimum to reduce network cost and processing overhead.
4. Operating under harsh conditions resulting in disconnected operation: Hostile phenomena like fires can lead to disruption of communications between the sensor network and the base station hence, the detection and tracking process should be able to operate in an autonomous manner without requiring remote supervision.

Representation

We represent a phenomenon cloud as a 5-tuple, $P = \langle a, b, p_T, m, n \rangle$. The lower and upper bounds of the range of sensor values which constitute a phenomenon are denoted by ‘a’ and ‘b’ respectively. For example, a hydrogen gas cloud can have ‘a’ = 20% volume and ‘b’ = 100 % volume. p_T is the threshold probability, m is the observation count and n is the minimum quorum. We refer to a sensor’s reading lying in the range [a, b] with probability greater than p_T during the last ‘m’ observations (that is, in a sliding window of size m) as satisfying the *Probability-Condition*. We define a sensor’s neighborhood as the set of sensors immediately

surrounding that sensor. We further state that a sensor is said to participate in a phenomenon cloud (or satisfy the *Phenomenon-Condition*) given by P = <a, b, p_T, m, n>, if it satisfies the *Probability-Condition* and at least 'n' sensors in its neighborhood also satisfy the *Probability-Condition*. This criterion ensures that a sensor must have a sufficient number of neighboring sensors in agreement with it before it can claim the existence of a phenomenon cloud, thereby reducing the occurrence of false positives. We define *Phenomenon-Set* to be the set of sensors satisfying the *Phenomenon-Condition*.

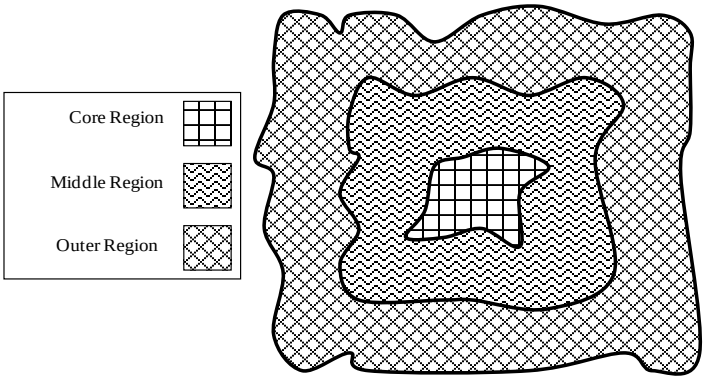

Figure 7-1. Dissection of a phenomenon cloud

We consider a phenomenon cloud to be composed of multiple regions (Figure 7-1). The innermost region or the Core region of the cloud is where the phenomenon is observed to be the strongest. The sensors lying in the Core region satisfy the requirements of the *Phenomenon-Condition* and hence, are members of the *Phenomenon-Set*. The Outer region denotes the fringes where uncertainty regarding the occurrence of the phenomenon is highest. The Middle region of the cloud is where the probability that the phenomenon is occurring is somewhere between that of the Core and Outer regions. The Middle and Outer regions are essentially areas where the occurrence of the phenomena has not yet been fully verified.

Detection and Tracking

In this section, we provide a detailed description of the phenomenon cloud detection and tracking process. We begin by classifying sensors into different categories according to the roles

they play in the detection and tracking process. Next, we describe the various responsibilities of a sensor node with respect to each of its neighbors, based on the categories they currently fall in. Then, we list a set of rules which govern the transition of sensors from one category to another and form a core part of our detection and tracking strategy. The rest of this section covers the different stages of the detection and tracking process in chronological order (Figure 7-3) and also describes mechanisms for handling node failures. Finally, we discuss how applications can utilize the real-time tracking data output by the sensor network.

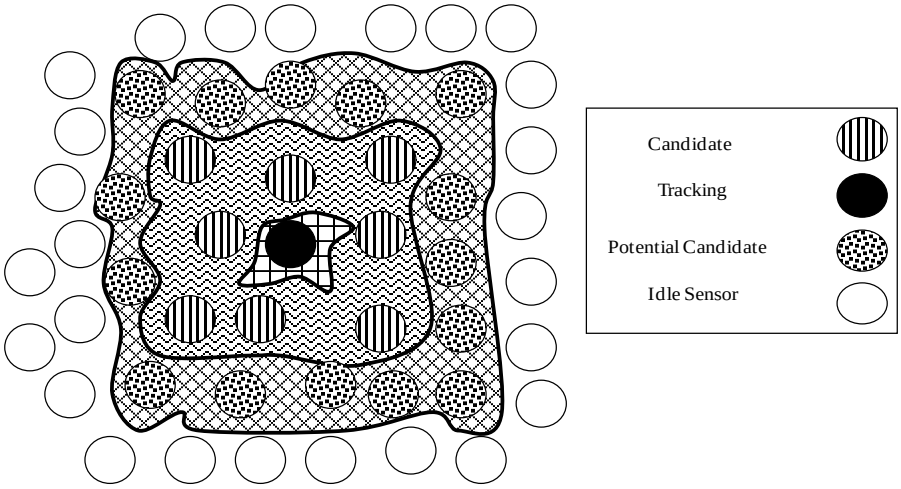

Figure 7-2. Classification of participating sensors

Classification of Sensors

Figure 7-2 superimposes the phenomenon cloud in Figure 7-1 over a group of sensors. Sensors are classified according to the region where they are located, which determines their role in the detection and tracking of a cloud. The different categories are

- **Candidate sensor**: A candidate sensor is one which is currently not part of any phenomenon cloud but is actively monitoring its readings to determine if it will become part of a cloud or not. Candidate sensors make up the Middle region of a phenomenon cloud. A sensor becomes a candidate if it has been selected as such by the query processor as part of the initial detection phase or it has transitioned from the potential candidate stage. The role of a candidate sensor is to receive notifications from its neighboring sensors whenever they satisfy the *Probability-Condition*. Based on the number of such neighbors it decides whether it is eligible to become a tracking sensor or not. Here we also note that in case a candidate sensor has one or more neighbors who are candidate sensors themselves, then it is responsible for notifying them whenever its own readings satisfy the *Probability-Condition*.

- **Potential candidate sensor**: All sensors which are immediate neighbors of candidate sensors but are not candidates themselves are called potential candidate sensors. These sensors also monitor their readings to enable a neighboring candidate sensor to check the validity of its observations. The role of a potential candidate sensor is to notify its neighboring candidate sensors whenever its readings satisfy the *Probability-Condition*. Potential candidate sensors form the fringes of detection and make up the Outer region of the phenomenon cloud. Essentially the set of potential candidate sensors forms a *phenomenon front* which grows and shrinks dynamically.
- **Tracking sensor**: A tracking sensor is a sensor which has already detected a phenomenon event and is now actively engaged in tracking it. A candidate sensor becomes a tracking sensor after it satisfies the *Phenomenon-Condition*. Tracking sensors make up the Core region. The *Phenomenon-Set* is the collection of all tracking sensors hence, each cloud that a user observes through the sensor network, consists of sub-sets of tracking sensors from the *Phenomenon-Set*.
- **Idle sensor**: All sensors which do not belong to any of the above three categories are called idle sensors. These sensors are not engaged in phenomenon detection or tracking and do not perform any monitoring whatsoever. Typically most sensors in the space will fall in this category since only selected clusters of sensors will be actively engaged in the detection and tracking of phenomena clouds at any given time. This ensures that the detection and tracking process is executed in a localized manner with minimal expenditure of energy.

Keeping Tabs on the Neighborhood

Each sensor node keeps track of which category each of its neighboring sensors falls in. This is done in a peer-to-peer fashion, where a sensor transitioning from one category to another, notifies the other sensors in its neighborhood via a 1-hop broadcast, without involvement of the centralized query processor. The Atlas nodes support ZigBee communication which natively supports 1-hop broadcasting. The categories a sensor and its neighbor fall into determine their mutual responsibilities towards each other. Table 7-1 lists the actions a sensor node is required to perform with respect to a neighborhood sensor, based on which category each of them belongs to. For example, a candidate sensor node A has a neighbor B which is also a candidate sensor and another neighbor C which is a tracking sensor. In that case, A will alert B whenever its readings satisfy the *Probability-Condition* but A will only alert C whenever its readings do not satisfy the *Probability-Condition*. Hence, a single sensor node can play different roles with respect to different neighbors depending on which categories they fall into. The cells marked

'Not Applicable' imply that such combinations are not possible according to transition rules given in the next subsection.

Table 7-1. Actions taken by a sensor node with respect to its neighbors which are not idle

Sensor node	Potential candidate	Candidate	Tracking
Idle	None	Not applicable	Not applicable
Potential candidate	None	Send alert whenever readings satisfy *Probability Condition*	Not applicable
Candidate	Receive alerts from neighbor whenever its readings satisfy *Probability Condition*	Send alerts/ receive alerts whenever their readings satisfy *Probability Condition*	Send alert whenever readings do not satisfy *Probability Condition*
Tracking	Not applicable	Send alert whenever readings satisfy *Probability Condition*	Send alerts/ receive alerts whenever their readings do not satisfy *Probability Condition*

Transition Rules

A set of rules govern the transition of a sensor from one category to another. These rules are executed in-network and control the entire detection and tracking process.

- **R1: Candidate→ Tracking** . If a sensor satisfies the *Phenomenon-Condition* then it transitions into the tracking category. Once a sensor is in the tracking category, it becomes a member of the *Phenomenon-Set*.

- **R2: Potential Candidate→ Candidate** . A potential candidate sensor will transition to a candidate sensor if any of its neighbors transitions into a tracking sensor. This rule corresponds to the fact that whenever a phenomenon cloud moves or expands and a new sensor becomes part of its core region, the phenomenon front also moves or expands.

- **R3: Idle → Potential Candidate**. An idle sensor transitions into a potential candidate if any of its neighbors becomes a candidate sensor.

- **R4: Tracking → Candidate**. A tracking sensor will transition down to the candidate category if it is unable to satisfy the *Phenomenon-Condition*. In such a case, the sensor will cease to be a member of the *Phenomenon-Set*.

- **R5: Candidate→ Potential Candidate**. A candidate sensor will transition to a potential candidate sensor if none of its neighbors are tracking sensors.
- **R6: Potential Candidate→ Idle** . A potential candidate transitions into an idle sensor if all its neighbors are either potential candidates or idle, that is none of its neighbors are in the candidate category.

Initial Selection of Candidate Sensors

The main goal of this stage is to detect initial occurrences of phenomena clouds. A phenomenon cloud can manifest itself in multiple locations simultaneously hence, monitoring one particular location is not adequate. However, in the interest of conserving network resources and power for the entire sensor grid, we cannot require each and every node to monitor its readings. A compromise between the two approaches can be followed where specific sensors are directly chosen to be candidates by the centralized query processor. The criterion for such a selection can be based on the location of nodes or past history of their readings. For example, if we are planning to detect gas leaks in a pipeline, it might be useful to choose the sensors located at the valves and joints to be the initial set of candidate sensors, since the probability of a leak getting started at those locations is higher. We make use of this criterion in the Smart Floor application described later, where sensors located near doorways are selected as initial candidates so that whenever a person enters the room, the system is immediately able to pick up their presence and commence the detection and tracking process to monitor their movement. Another criterion can be the offline use of an available mathematical model of the phenomenon cloud to determine locations where the probability of occurrence is the highest. In case such a criterion is hard to formulate, alternatively the system can randomly select sensors as candidates such that they are uniformly distributed over the sensor space. These sensors and their respective neighborhoods can be viewed as autonomous clusters of early warning systems for detecting the sudden manifestation of possibly multiple phenomena clouds.

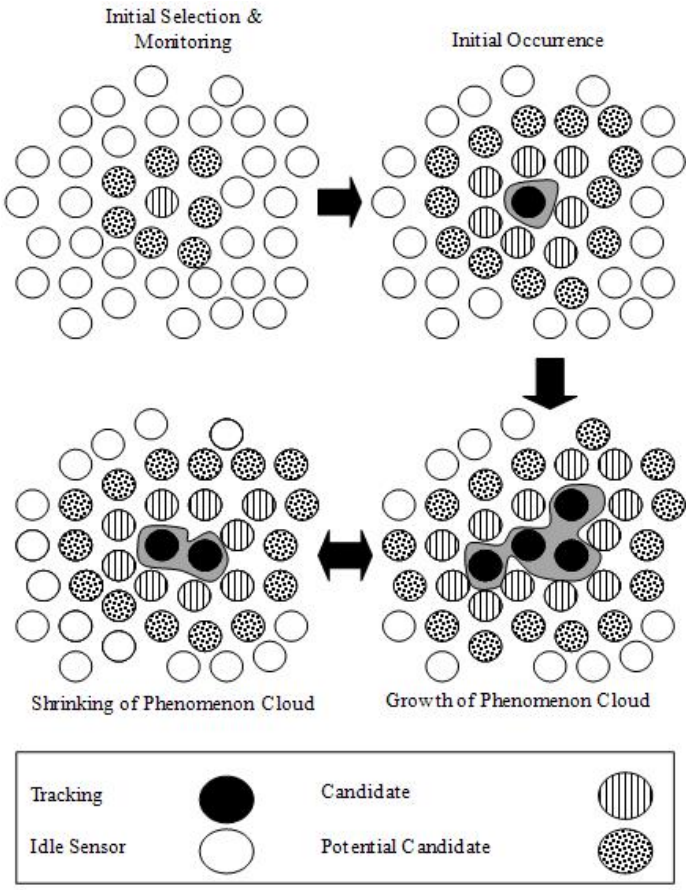

Figure 7-3. Detection and tracking of a phenomenon cloud

Since sensor deployment patterns tend to be highly application and phenomena-specific, we do not go into details of their deployment. For purposes of discussion for the rest of this chapter, we assume that sensor nodes are deployed in such a manner that each sensor has a sufficient number of neighbors to potentially avoid false positives.

Monitoring for Initial Occurrences

The query processor pushes the phenomenon cloud parameters on to each of the selected initial candidate sensor nodes in the network. At the beginning of every epoch, each potential candidate node monitors its readings and sends a 1-hop broadcast message every time the *Probability-Condition* is satisfied. In order to enable sensor nodes to receive alert broadcasts from multiple neighbors simultaneously during the same epoch, a slotted approach is used to

ensure collision avoidance. Each epoch is sub-divided into multiple sub-epochs and each node only broadcasts alerts during its assigned sub-epoch. The candidate sensor node aggregates alerts received via broadcasts from its neighbors and determines if it satisfies the *Phenomenon-Condition*. A candidate sensor satisfies the *Phenomenon-Condition* if its readings satisfy the *Probability-Condition* and it also receives broadcast alerts from at least 'n' neighbors which have also satisfied the *Probability-Condition* in the same epoch.

Notification of Initial Occurrence

If a candidate node has satisfied the *Phenomenon-Condition*, it notifies the query processor residing in the base station that it has detected presence of a phenomenon cloud. The query processor adds the candidate node to the *Phenomenon-Set* and the candidate sensor transitions to a tracking sensor role using rule R1.

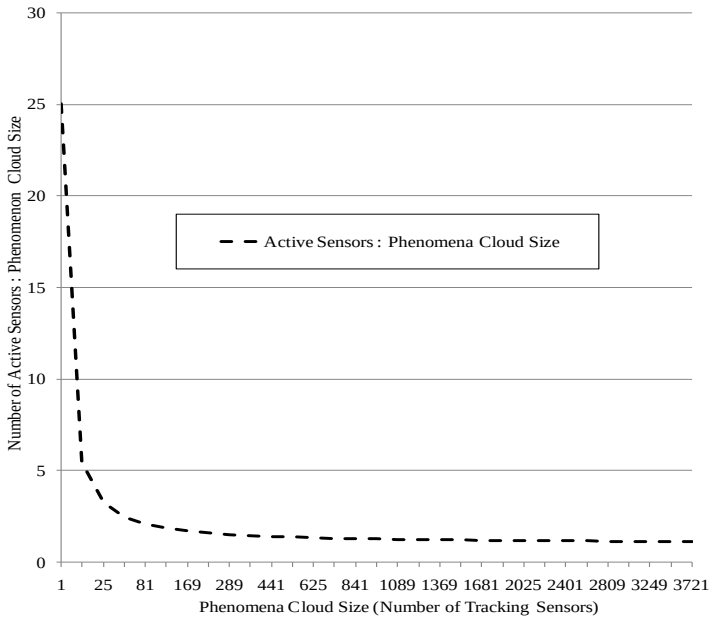

Figure 7-4. Ratio of total active sensors to cloud size in a rectangular sensor grid

Growth of Phenomenon Cloud

The candidate sensor also broadcasts an alert about its change of state. As soon as the neighborhood sensors receive the alert, they transition to the candidate category using rule R2.

Each of these sensors in turn, broadcasts its change of state and this causes all their respective neighbors which had been idle till now, to transition into potential candidate category (rule R3). In this manner, the detection mechanism gets distributed and propagated in-network, without involvement of the centralized query processor, as the phenomenon cloud grows with time. Each sensor node keeps track of its neighborhood via the broadcast alerts that it receives and determines the actions to be undertaken with respect to a specific neighbor based on which category each neighbor falls in (Table 7-1).

We observe that in our distributed in-network approach, the number of active sensors required at any given time is only slightly more than the number of sensors actually participating in the phenomenon cloud and the ratio of active sensors versus phenomena cloud size decreases with increase in cloud size (Figure 7-4). This is due to the fact that the detection and tracking process is executed in-network in a localized manner to ensure maximum efficiency. Only those sensor nodes which are in the immediate vicinity of a phenomenon cloud or are participating in the cloud are actively involved in the detection and tracking process. The propagation of this process in the network is governed solely by the behavior of the phenomenon cloud and handled by the distributed nodes in a co-operative manner using the rules (R1 – R6) specified earlier, without requiring assistance from the centralized query processor.

Shrinking of Phenomenon Cloud

The phenomenon cloud is said to shrink when sensors falling in the tracking category determine that they can no longer satisfy the *Phenomenon-Condition*. After a sensor transitions into tracking, its neighbors will only send it alerts if their readings fail to satisfy the *Probability-Condition* (Table 7-1). A tracking sensor is no longer participating in the observation of the phenomenon cloud if it determines that less than ‘n’ of its neighbors currently satisfy the *Probability-Condition*. In such a case, the tracking sensor node notifies the query processor

which removes the tracking sensor from the *Phenomenon-Set* thereby signifying that the phenomenon cloud has shrunk. The affected sensor also transitions into the candidate category using rule R4 and notifies the other sensors in its neighborhood of this role change via broadcast. The neighbors which are candidate sensors and meet the conditions of rule R5 then transition into potential candidate sensors. Furthermore, the neighbors of these sensors which meet the conditions of rule R6 transition into idle sensors. We make a note that if all the phenomena clouds disappear completely then after all the transitions are applied as per the given rules, the sensor space will revert back to the initial state, where only the initial set of candidate sensors will remain active along with their respective neighborhoods consisting of potential candidate sensors.

Handling Failures

The failure of sensors is an inevitable part of deploying sensor networks. In the context of phenomenon detection and tracking, this is especially true for phenomenon clouds which occur in hostile conditions such as wild fires or occur in very large scale sensor deployments. In case ZigBee communication is used, a sensor node can detect failure among its neighbors while notifying them to transition into a different category. ZigBee networks implement ACKs on the MAC level, hence, if any of the neighbors are dead the node will be able to detect that its messages could not be delivered to a particular neighbor successfully. Whenever a node detects that one of its neighbors is dead, it updates its total neighbor count (call it N). We note that every node at some point in time, as per the terms of the *Phenomenon-Condition* and rules R1 through R6, will transition to a different category either due to its ability or inability to satisfy the *Phenomenon-Condition* or based on the transition of its neighbors. Hence, whenever the node to inform its neighbors of this transition, it will be able to detect which of them are dead. A more robust but slightly more expensive way to ensure which nodes in a neighborhood are alive is to

require each node to broadcast 'alive' messages at regular intervals if it has not broadcast any alerts for a certain period of time.

In case a candidate sensor fails, all of its neighbors transition into candidate roles on detecting the failure. Also, if the value of a node's neighbor count N falls below a certain minimum value N' ($\geq$ n), it alerts its neighbors via 1 -hop broadcast so that potential candidates among them can transition into candidate category, and their idle neighbors in turn can transition into potential candidate category. This strategy ensures that the detection of the phenomenon cloud does not get stalled due to failure of any of the candidate sensors or lack of enough sensors in a candidate node's neighborhood.

Real-Time Monitoring by Applications

The centralized query processor maintains the *Phenomenon-Set* which as described before, is the collection of all sensor nodes which fall into the tracking category at any given time. The query processor is able to track phenomenon clouds in real-time, with minimum processing and networking overhead of receiving updates from the sensor nodes. The *Phenomenon-Set* is only updated whenever a sensor transitions to or from the tracking category. Hence, the query processor only requires minimal updates to continuously track phenomena clouds.

Since the location of each sensor node is known beforehand, an application such as a phenomenon cloud visualization tool with a graphical user interface (GUI) can easily reconstruct a view of the various phenomenon clouds in real-time using information from the *Phenomenon-Set* in conjunction with sensor location information. By looking at which sensors enter or leave the *Phenomenon-Set*, the motion of multiple phenomenon clouds can also be tracked over time and more sophisticated analysis and prediction performed at a centralized level. This is extremely useful for applications such as those which can determine safe passages for rescue workers through multiple occurrences of phenomenon clouds such as gas leaks and wild fires

[4]. As future work, we are concentrating on better organization of the *Phenomenon-Set* to enable efficient updates and look-ups of specific clouds and their characteristics. The cardinality of the *Phenomenon-Set* in combination with sensor location can give applications information about the size of various phenomena clouds. The size of a phenomenon cloud can have different impacts depending on the application context. For example, if an application is concerned with detection of phosphate dust clouds, a *Phenomenon-Set* of low cardinality may not have much significance. However, if an application is tasked with the detection of hydrogen cyanide (HCN) leakage, then the detection of even a small cloud indicates serious consequences.

Figure 7-5. Gator Tech Smart House

A Practical Application of Phenomena Detection and Tracking

We describe a real-world application of phenomena detection and tracking which is radically different from the gas cloud and oil slick simulations that are usually presented. The application that we describe here involves the Smart Floor [6, 17] in the Gator Tech Smart House [13] (Figure 7-5). The Smart Floor deployed in 2005, consists of a grid of force sensors deployed under the raised floor tiles of the house. Each tile has a single sensor connected to an Atlas Platform node [18] placed below its center (Figure 7-6), which allows a step anywhere on the tile to be detected. The Smart Floor covers the entire residential area of the 2500 sq. ft. house and

allows it to monitor its residents' movement and location without encumbering them with tags or other tracking devices.

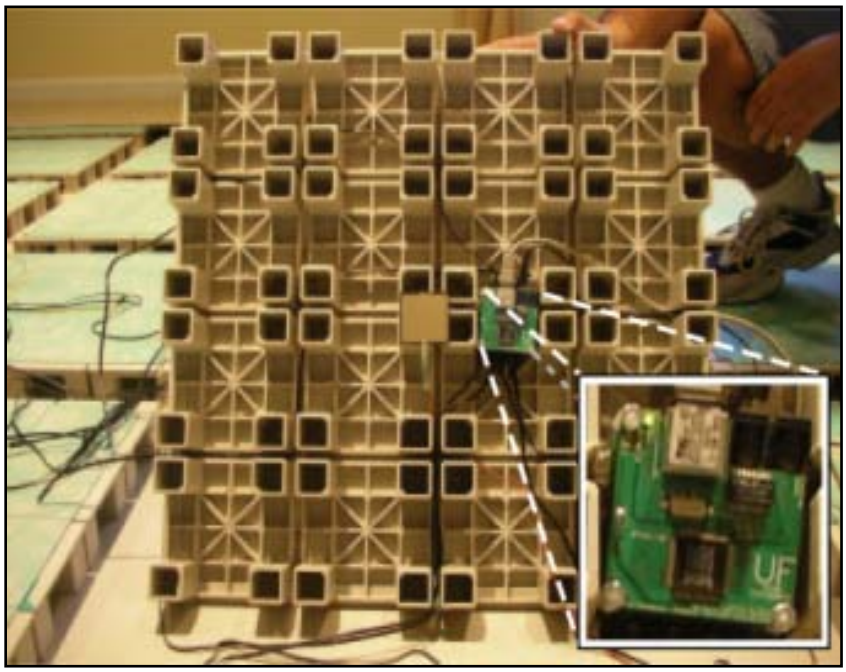

Figure 7-6. Smart Floor tile with force sensor and Atlas Platform node

While designing the Smart Floor application, the naïve expectation was that when a person steps on a tile, only the sensor underneath that tile outputs a reading of significant magnitude. Unfortunately, based on our experience over the years we found that this was clearly not the case. Due to various reasons including seemingly random vibrations, individual sensors sometimes output large readings even when nobody is stepping on them. This results in a very noisy sensory environment where one cannot distinguish between a genuine step and random spikes by relying on individual sensors alone.

We also observed that when a person steps on a tile not only does this result in that tile's sensor registering a strong reading but some of its neighboring tiles also output significantly large readings. Hence, the stepping action of a foot on a floor tile causes a ripple effect in the immediate neighborhood of the tile (Figure 7-7) as seen in an actual screenshot of this

phenomenon occurring in the Smart Floor where red dots indicate tiles registering readings of higher magnitude and green indicates tiles with lower yet significant magnitude. We used this observation to describe walking as a phenomenon by defining a step in terms of a phenomenon cloud (Figure 7-8), in order to reduce the number of false positives and provide accurate location information about the home's resident. Moreover, since our approach to phenomenon detection and tracking does not rely on mathematical modeling to track the direction of movement of a phenomenon hence, this makes it extremely suitable for observing phenomena such as walking where it is difficult to accurately model the path that a person will follow at any given time.

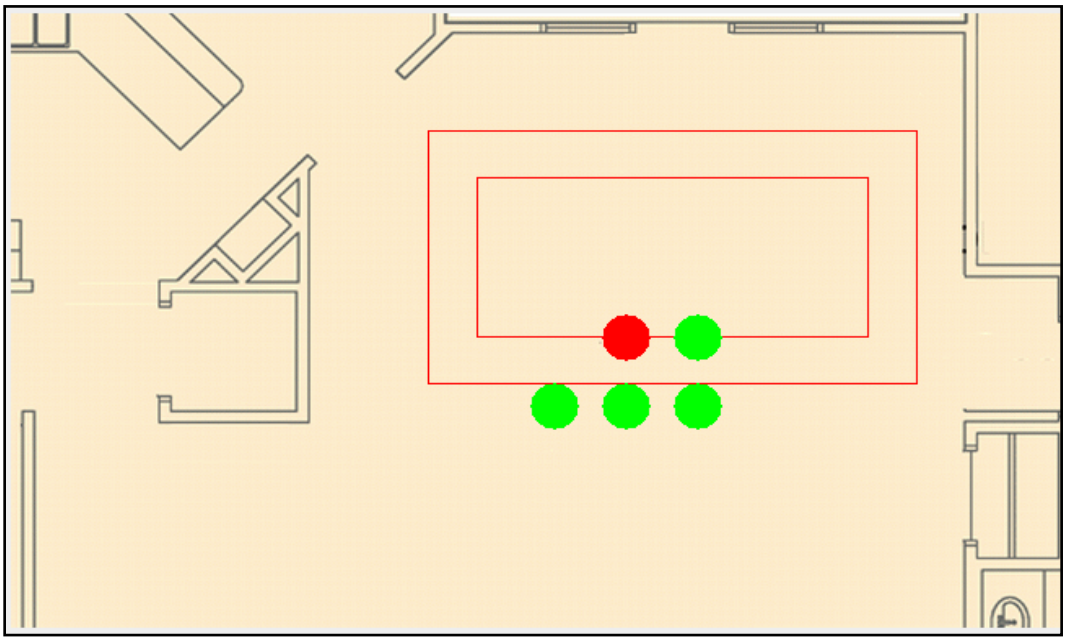

Figure 7-7. Ripple effect of a foot step on the Smart Floor

A step can be described as a phenomenon cloud S = <a, b, p_T, m, n>, where 'a' and 'b' denote the lower and upper bounds of a force sensor reading indicating that a foot has stepped on a tile or in its immediate vicinity. This value depends on the particular sensor being used. For example, based on empirical study, we found that for the Interlink force sensors used in the Smart Floor (having an output range of [0, 1023]), 'a' is equal to 150 and 'b' is equal to 600 for

an individual weighing between 110 to 240 pounds. The optimal values of the other parameters were determined via experimentation and are described in the following section.

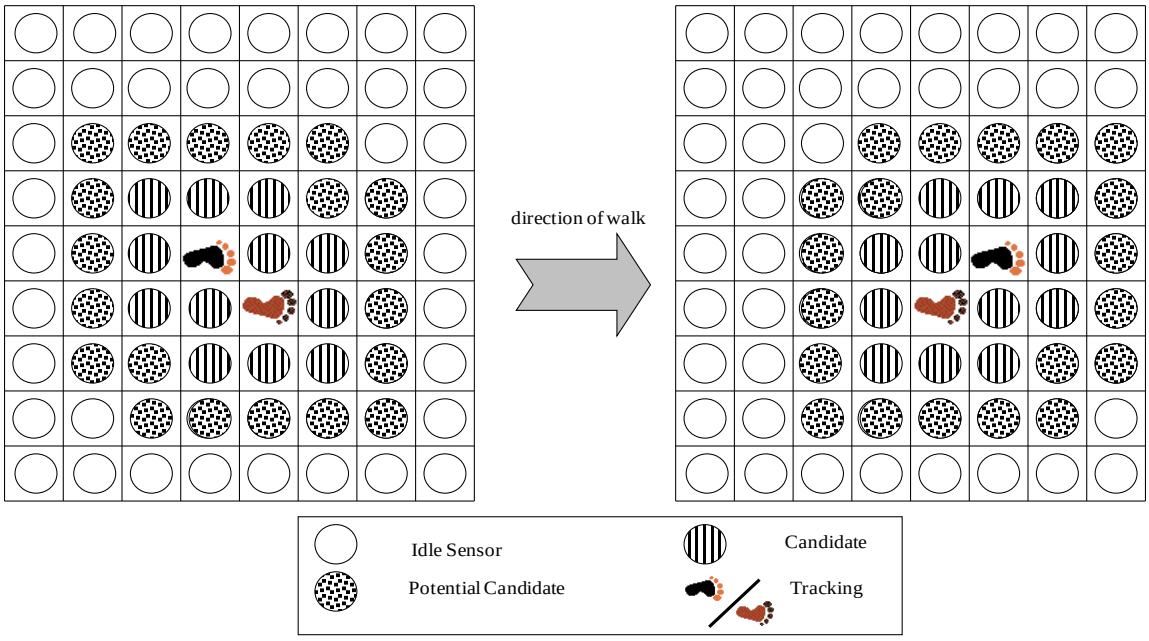

Figure 7-8. Walking motion as a phenomenon

Walking motion is characterized by stride length (the distance between two footfalls of the same foot), gait velocity (speed at which a person walks) and cadence (the number of steps a person takes per minute). The observation of these parameters is of paramount importance for monitoring patients suffering from obesity and Parkinson's disease. For example, people with Parkinson's disease have significantly shorter stride length and slower gait velocity as compared to healthy individuals. Similarly, people suffering from morbid obesity typically have comparatively low gait velocity and cadence. Phenomena detection and tracking can be used to monitor all three walking parameters in the privacy of one's home without encumbering the resident in any way. Stride length can be instantaneously calculated as twice the distance between two sensors which send consecutive update messages. Gait velocity can be calculated over an observation period P whose length depends on how long the resident walks in a straight path without turning. If tiles 'i' and 'j' are the first and last tiles stepped on during P, then gait

velocity can be calculated as the difference between $distance_i$ and $distance_j$ divided by the magnitude of P. Finally, cadence can be calculated as one-half of the number of update messages received by the monitoring application in one minute.

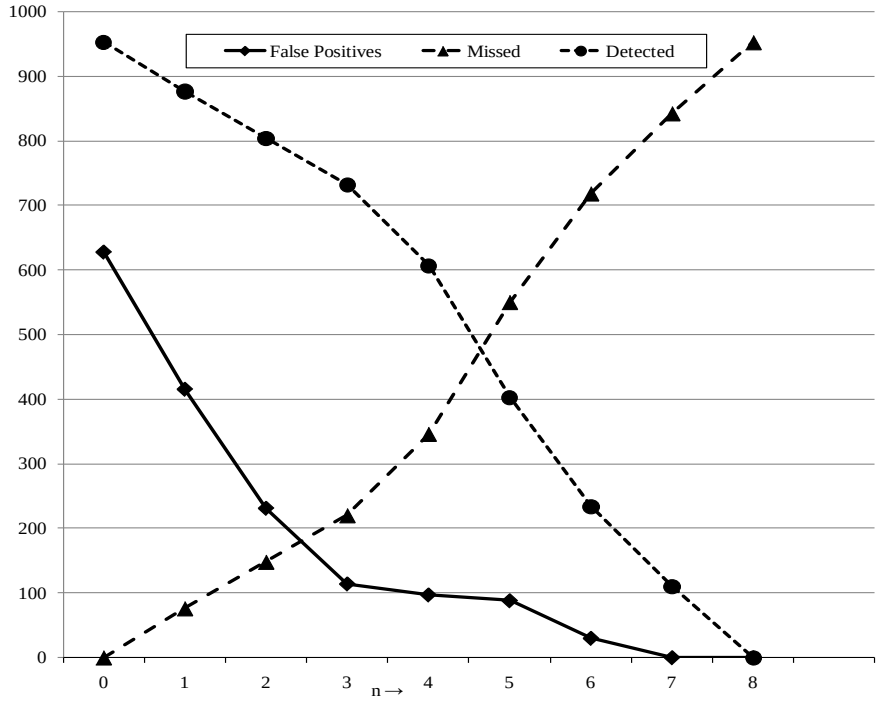

Figure 7-9. Effect of varying 'n' with p_T=0.4 and m=150

Experimental Analysis and Performance Evaluation

We evaluated various aspects of the distributed phenomenon detection and tracking approach using both real-world and simulation experiments. The first set of experiments evaluated the effectiveness of our detection strategy in a real world sensor deployment and analyzed the effect of the varying phenomena definition parameters 'p_T', 'm' and 'n'. The second set of experiments used simulation to evaluate the resource usage and energy consumption of our approach and compare it with that of stream-based detection and tracking. We relied on simulation in this case because we wanted to measure resource usage and power consumption in large sensor networks of varying sizes. Hence, it was not practically feasible for us to physically deploy sensors in such large numbers for the purpose of experimentation.

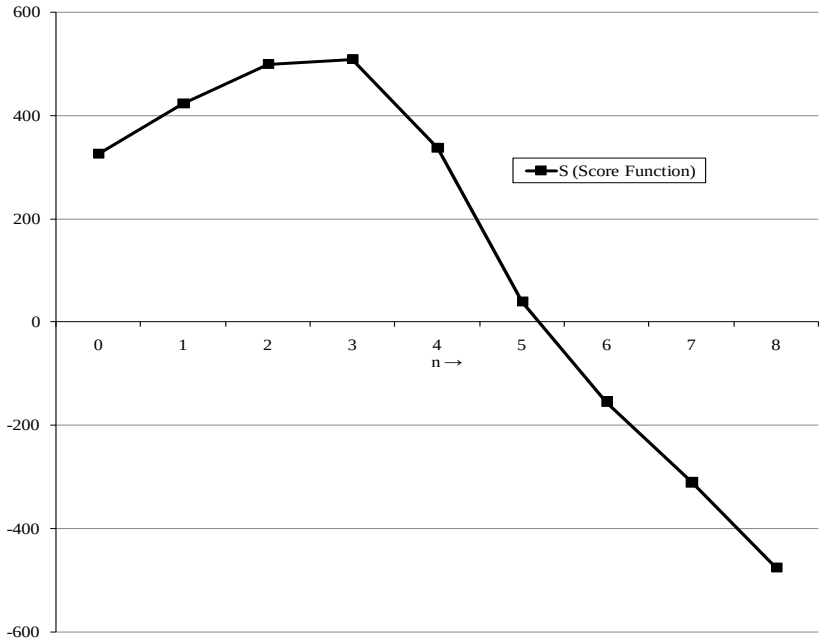

Figure 7-10. Determining the optimal value of 'n'

Experiment I: Effectiveness of Detection Strategy

In this first set of experiments we study the effectiveness of our phenomenon detection and tracking mechanism in a real-world sensor deployment inside the Gator Tech Smart House.

Experimental setup

We chose to evaluate effectiveness by performing experiments using the Smart Floor in the Gator Tech Smart House, where human footsteps are represented as phenomena clouds and phenomenon detection and tracking is used to monitor the location of the resident in the house. We observed the effect of phenomenon definition parameters 'p_T', 'm' and 'n', on the detection efficiency of our approach. We varied the values of each of these parameters and studied their effect by logging the number of false positives, false negatives/misses and correct detections of a human step. In order to aid our evaluation, we restricted movement to a 100 sq. ft. area in the living room of the smart house and had test subjects walk along a clearly marked path on the floor. This allowed us to log the steps that a person was taking and collect statistics on correct detections and detection errors.

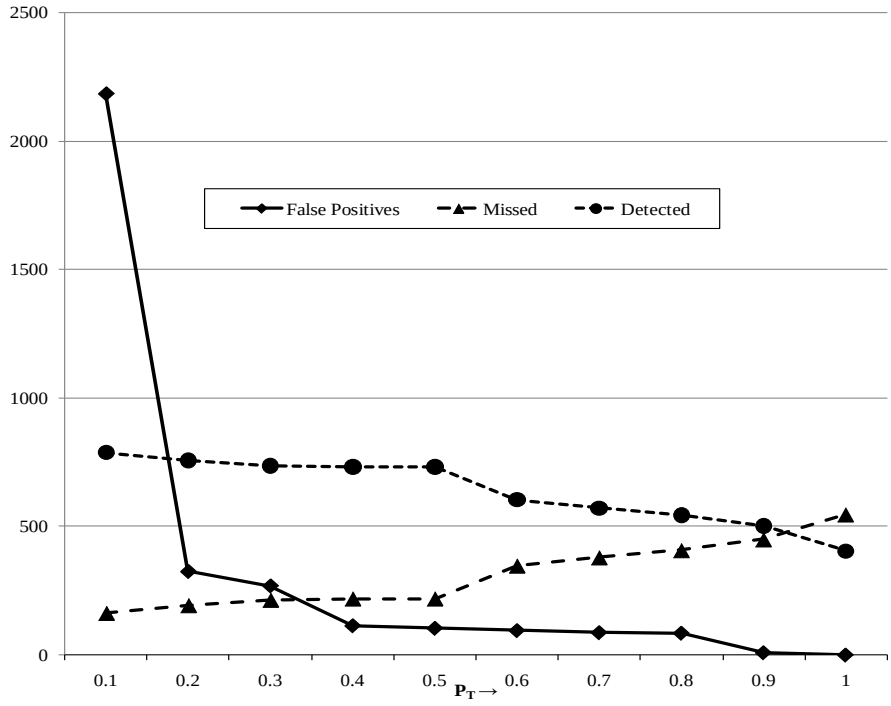

Figure 7-11. Effect of varying 'p_T' with n=3 and m=150

Results and analysis

The experimental results are presented in three graphs (Figures 7-9, 7-11 and 7-13). First we consider the effect of varying parameter 'n' (Figure 7-9) which determines the minimum quorum of neighboring sensors required to conclusively determine the occurrence of a phenomenon. Since the Smart Floor is deployed as a rectangular grid of sensors, the value of 'n' varies from 0 to 8. We observe that for n=0 (which corresponds to the naïve case) the number of false positives is extremely high since the system is entirely relying on outputs from single sensors to determine the occurrence of a footstep event. Hence, even though there are no misses/false negatives and all the actual footsteps are detected, their occurrence is lost in the noise of having an extremely large number of false alerts. As we increase the value of 'n', the number of false positives comes down sharply since now multiple neighboring sensors need to agree on the occurrence of a phenomenon. We also notice that as 'n' increases, the number of misses also increase, thereby reducing the number of correct detections. This is due to the fact

that walking is essentially a transient event where a footstep has to be detected by a specific sensor within a very small time window. Hence, even though we postulated that the action of stepping on a tile causes a ripple effect among neighboring tiles, it is not necessary that the number of neighbors experiencing this effect will always meet the minimum quorum requirement ('n') within the time window. For large values of 'n', the number of misses is very high and consequently, the number of correct detections becomes very low, since the quorum requirements become too stringent and cannot be satisfied in most or all cases. In order to determine the optimal value of 'n', we devised a score function (Equation 7-1) which is a weighted sum of the number of successful detections, number of false positives and number of false negatives.

$$S(D, FP, FN) = D - FP - 0.5FN \quad (7\text{-}1)$$

where D is the number of footsteps detected successfully,
FP is the number of false positives and
FN is the number of false negatives.

In case of the Smart Floor, we decided that the occurrence of a false positive is less desirable than the occurrence of a false negative and we set their respective weights in the equation accordingly. Note that the formula of the score function (Equation 7-1) is specific to the Smart Floor and it is dependent on the type of application and the relative impact of false positives and false negatives on it. We found that for the Smart Floor in the Gator Tech Smart House, the score is the highest when n is equal to 3 (Figure 7-10). This is supported by the fact that this value of n ensures a reasonably good level of performance, where the number of false positives is comparatively low as compared to the number of correct detections and approximately 77% of all footsteps are successfully detected.

We observe that as threshold probability p_T increases, the number of false positives decreases since it filters out random spikes (Figure 7-11). Random spikes typically result in only

a few readings of significant magnitude within a fixed size sliding window hence, there is a sharp drop in the number of false positives even when we only increase p_T from 0.1 to 0.2. However making the probability requirement more stringent also results in an increase in the number of false negatives/misses. This is due to the fact that we are using a sliding window of fixed size. As the requirement on the number of readings that have to lie within the phenomena-defined bounds [a, b] increases, the chances of the *Probability-Condition* getting satisfied decreases. Using the score function defined previously (Equation 7-1), we found that setting p_T equal to 0.4 (Figure 7-12) results in a reasonably good detection rate with a low number of false positives and misses.

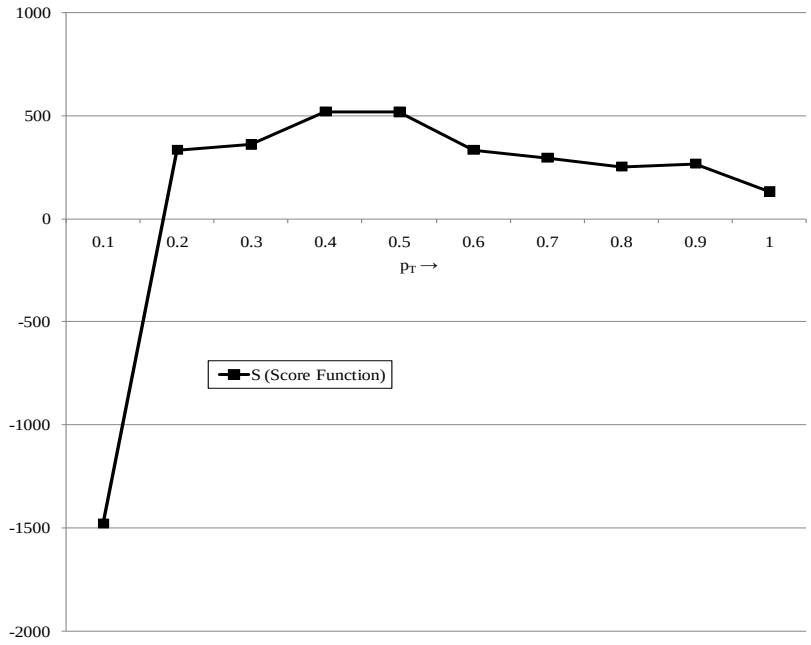

Figure 7-12. Determining the optimal value of 'p_T'

We observe that if the sliding window size is too low, this results in a large number of false positives despite having a high probability threshold (Figure 7-13). This is due to the fact that in case of sensors which have a high sampling rate even random spikes can result in a fairly large contiguous set of significant readings. Since the system essentially uses the threshold frequency ($m.p_T$) to evaluate whether a sensor satisfies the *Probability-Condition*, for small window sizes

the corresponding threshold frequency is also low even if threshold probability p_T is kept high. Hence, there is a high probability that random spikes get mistaken for actual footsteps. Increasing the sliding window size on the other hand, raises the threshold frequency, which as we can observe results in a moderate increase in the number of misses. Using the score function (Equation 7-1), we found that for the Smart Floor, setting the sliding window size m=150 results in reasonably good detection performance without taxing memory resources of individual sensor nodes (Figure 7-14).

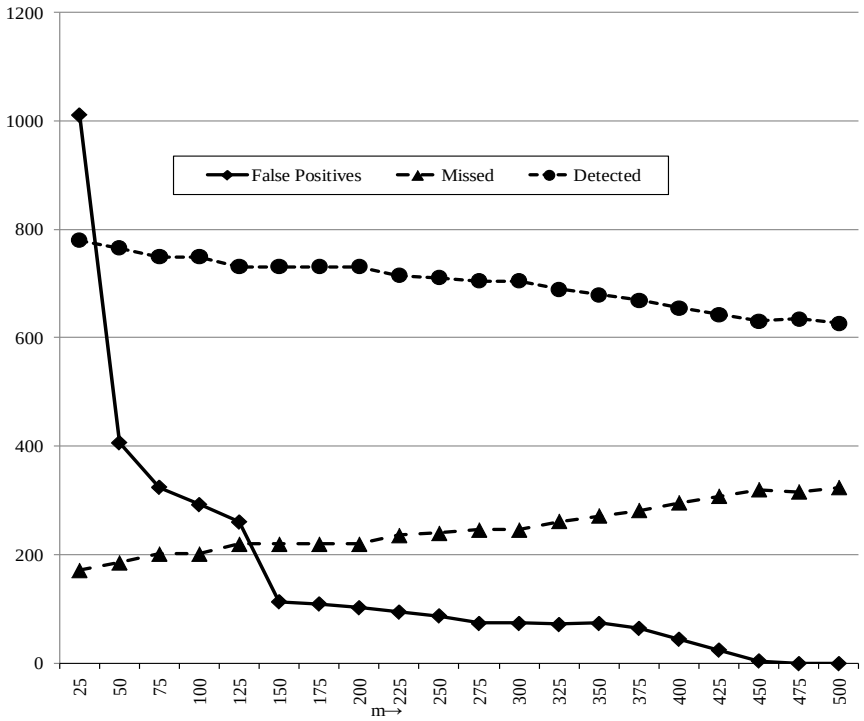

Figure 7-13. Effect of varying 'm' with n=3 and p_T=0.4

Experiment II: Resource and Power Consumption

In the second set of experiments we studied the system resource and energy consumption characteristics of our approach and compared it with that of a stream-based approach. We used a simulation-based approach where we simulated the spawning and random movement of multiple phenomenon clouds on a rectangular grid of sensors and conducted a comparative analysis of the resource and energy consumption of each technique.

Table 7-2. Energy consumption specifications for Atlas ZigBee nodes

Operation	Current (mA)	Duration (seconds)
Sampling Sensors, Processing or Listening for messages	6.0	2.0
Receive Message @256kbps	15	0.002
Transmit Message @256 kbps	15	0.002

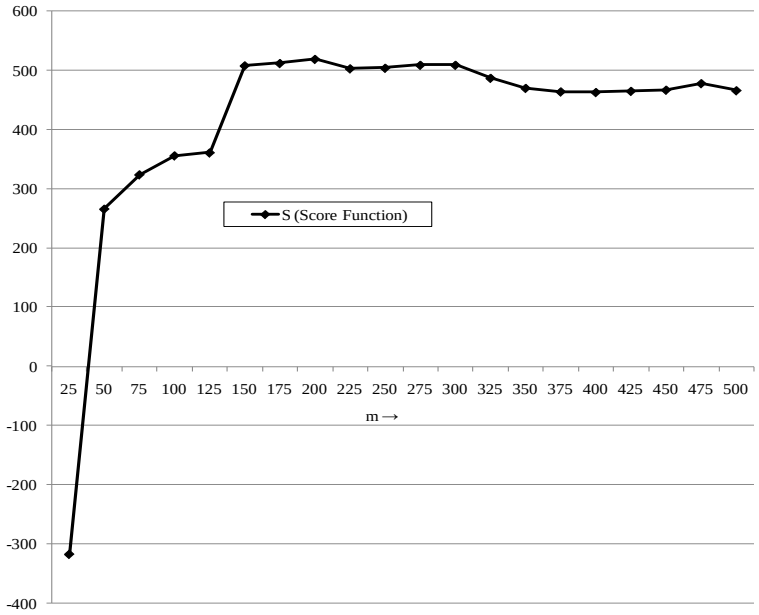

Figure 7-14. Determining optimal value of 'm'

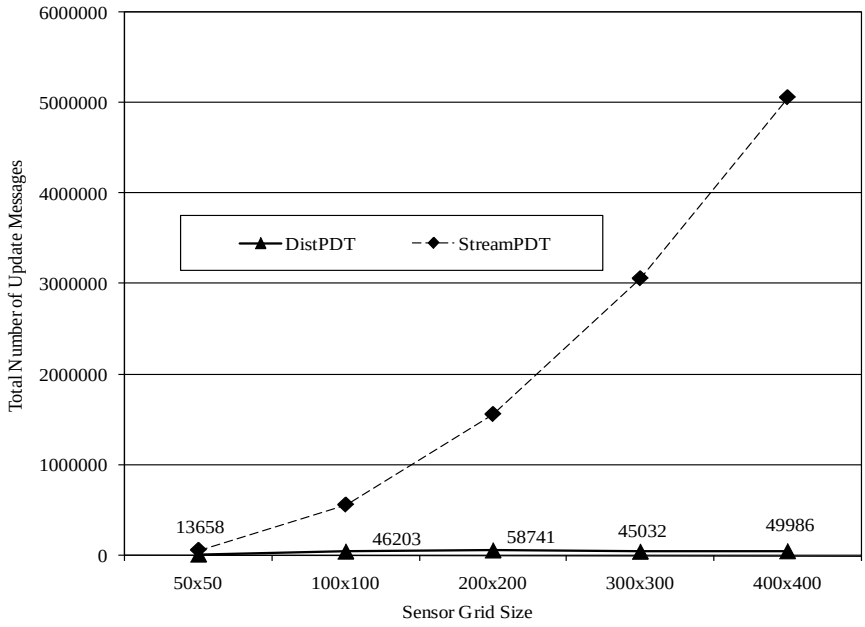

Figure 7-15. Number of update messages sent to the query processor

Experimental setup

We simulated the movement of phenomenon clouds over 100 epochs on a rectangular grid of sensors of sizes varying from a 50 x 50 grid to a 400 x 400 grid. Each sensor was connected to an Atlas ZigBee node. At the beginning of each simulation, between 2 to 5 phenomenon clouds were spawned randomly in different areas of the grid. During each epoch, the direction of motion for each cloud was randomly decided and the variation of its motion and size simulated by selecting subsets of sensors which entered or left the *Phenomenon-Set*. Hence, the simulation can be viewed as a random walk of multiple phenomenon clouds over a sensor grid. Our simulation introduced a high level of uncertainty regarding phenomenon cloud movement and tested the performance of detection and tracking algorithms to the fullest extent, especially in the presence of multiple clouds. Statistics such as the number of network messages, processing costs and updates sent to the query processor were logged during the simulation process. The energy consumption of the nodes was calculated as a function of processing costs (including sampling sensors) and network costs (incurred in receiving and transmitting data over the radio) and was based on the Atlas ZigBee node hardware specifications (Table 7-2), which is in turn based on the Atmel Zlink Radio Control Board (RCB) design.

The detection and tracking strategies that were simulated in the experiments for purposes of comparison were: (1) StreamPDT which is the centralized stream-based algorithm where phenomena detection and tracking is performed by a centralized query processor such as Nile-PDT and; (2) DistPDT, the distributed acquisition-based phenomenon detection and tracking algorithm described in this chapter.

Results and analysis

The simulation results are shown in Figures 7-15, 7-16, 7-17 and 7-18. The numbers shown on each graph correspond to values of data points of their respective plots for DistPDT.

First, we consider the total number of updates regarding the position of phenomena clouds that were sent to the query processor during the entire course of the simulation. The total number of update messages sent is a reflection of the processing and networking workload put on the centralized query processor by each algorithm and can be considered to be a comparative test of scalability.

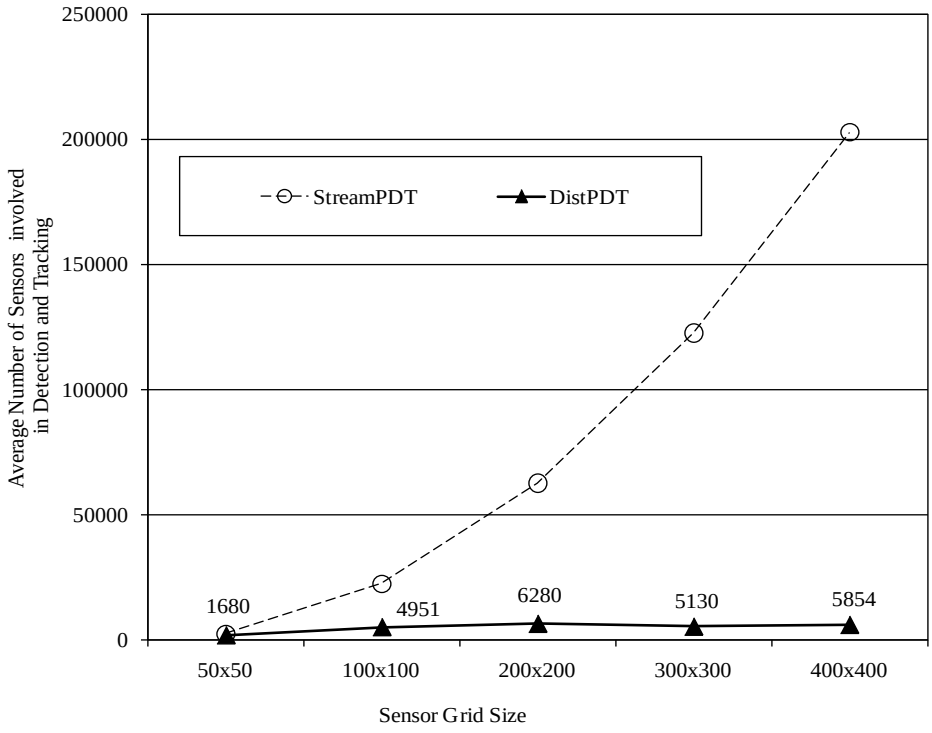

Figure 7-16. Average number of active sensors required

We observe that in the case of StreamPDT the number of update messages increases significantly with the size of the sensor grid (Figure 7-15). This is due to the fact that stream-based algorithms use centralized processing as opposed to in-network processing. Therefore, these mechanisms require inputs from all the sensors in the grid to detect and track phenomenon clouds. The number of update messages sent by DistPDT shows negligible increase as sensor grid size varies. In fact, DistPDT requires transmission of between 78% to 99% fewer update messages than StreamPDT This can be explained by the fact that in DistPDT, all the processing

is done in-network and update messages are only sent to the query processor when a phenomenon cloud grows, shrinks or moves that is, when sensors move in or out of the *Phenomenon-Set*. Therefore, regardless of the increase in size of the sensor grid, given similar phenomenon cloud patterns, the number of updates required by the query processor from the DistPDT mechanism does not change significantly, thereby reducing the query processor's work load and avoiding potential bottlenecks.

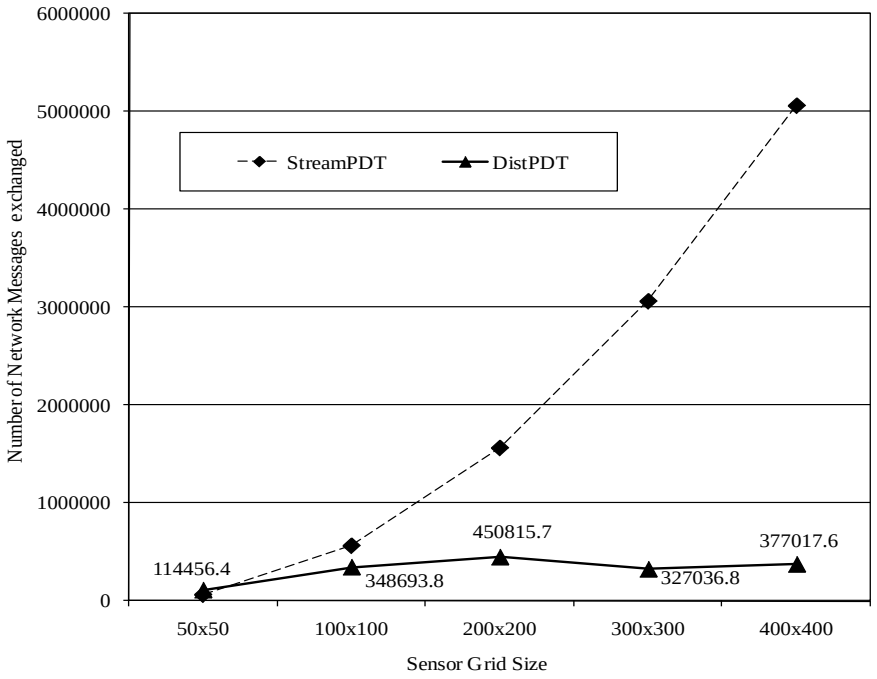

Figure 7-17. Number of network messages exchanged

This becomes an important issue when we consider smart spaces where base station hardware is not a full-fledged computer or server but instead might be something along the lines of a set-top box with limited processing and memory resources. In that case the base station will lack the high-performance computing capabilities required to simultaneously process data streams from a large number of sensors. In such circumstances our approach is more suitable since the query processor running inside the set-top box is only required to inject queries into the sensor network and receive intermittent updates about the state of the phenomena in the smart

space. The actual execution of the detection and tracking process is done in-network in an autonomous fashion without requiring any form of intervention from the centralized base station.

Next we look at the average number of sensors involved in the real-time tracking of phenomenon clouds during simulation (Figure 7-16). Since StreamPDT requires inputs from all sensors in the grid to be streamed up to the query processor, the increase in number of sensors actively involved in sampling their readings is equivalent to the increase in sensor grid size. For DistPDT, we observe that the average number of active sensors is around 2000-6000 sensors, anywhere from 30% to 98% less than StreamPDT. This is due to the fact that, the DistPDT mechanism tries to localize the process of detection and tracking to areas of the grid, where phenomena clouds have been observed or are likely to be present, significantly reducing the number of sensors which have to be actively sampled. Moreover, due to localization of in-network processing among sensor nodes, the number of active sensors does not change significantly with increase in grid size, given similar phenomenon cloud behavior. Distributed, in-network detection and tracking reduces bottlenecks but this comes at the cost of exchanging a higher number of inter-node messages to coordinate efforts among multiple nodes. Unlike StreamPDT where coordination is done in a centralized fashion and data aggregated at a single sink, DistPDT requires each sensor node to co-ordinate its own activities based on the state of its neighboring sensors. This naturally leads to an increase in the total number of network messages exchanged as compared to centralized mechanisms. However, this is true only for small grid sizes, where the percentage of sensors participating in the detection and tracking process will be higher than larger grids (Figure 7-17). For large grids, the cost of additional inter-node messaging incurred by DistPDT is comparatively outweighed by StreamPDT's processing and multi-hop networking costs incurred by non-participating sensor nodes.

If we consider the average energy consumption per node over the entire simulation (Figure 7-18), we observe that DistPDT has 60% to 98% lower average energy consumption than StreamPDT. Hence, DistPDT enables non-participating sensors to conserve their energy and prolong their availability for other tasks. Based on these experiments, we can conclude that DistPDT is much more scalable and energy-efficient than centralized stream-based algorithms, thereby making it suitable for practical deployment in real-world smart spaces such as the Gator Tech Smart House.

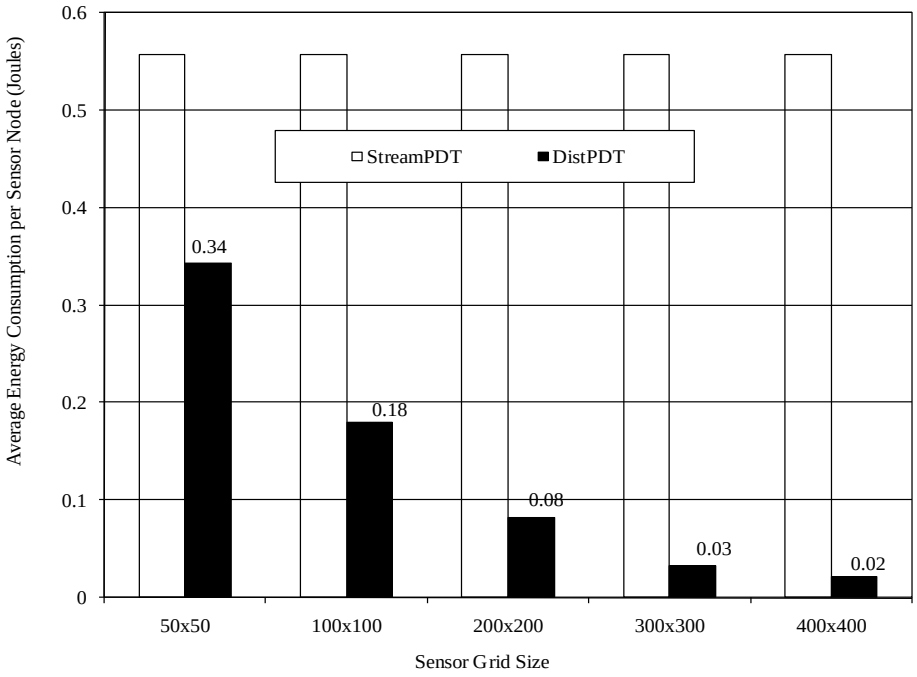

Figure 7-18. Average energy consumption per node

CHAPTER 8
CONCLUSIONS AND FUTURE WORK

We described *Sensable*, a scalable query processing middleware for service-oriented sensor networks. *Sensable* leverages service oriented architecture (SOA) concepts to provide query and event processing capabilities which go beyond the primitive types of data directly available from individual hardware sensors and allow the sensor network to handle queries involving abstract data types and difficult to define events and phenomena. It also utilizes knowledge associated with devices to minimize energy consumption of sensor networks utilizing new low-power networking technologies such as ZigBee.

Future work for the *Sensable* project is as follows. For sensor-aware query processing, we are planning on conducting experimental evaluations of query plan performance based on actual execution monitored on various sensor platforms connected to real sensors. We are also considering more rigorous simulation-based evaluation of plan performances using specifications of other contemporary sensor platform hardware. Furthermore, we are exploring more sophisticated methods for optimizing sensor sampling schedules across the network, especially approximation techniques based on model-based prediction.

For the virtual sensors framework, we are planning on developing fault tolerance mechanisms for derived virtual sensors by using the SOA-based virtual sensor concept in conjunction with service selection and replacement mechanisms for ubiquitous service composition in pervasive spaces. This implies that whenever a virtual sensor fails, service selection and replacement mechanisms will be activated to locate another suitable virtual sensor or instantiate a new one. We are also actively pursuing real-world virtual sensor application deployments in the Gator Tech Smart House and are currently concentrating on applying virtual sensors for unencumbered monitoring of conditions such as obesity and diabetes of its residents.

Finally, in the case of phenomena detection and tracking, we are looking at model-assisted detection and tracking where, distribution of detection tasks will be streamlined based on predictions regarding direction of movement of phenomenon clouds. We do not require the existence of formal cloud models for our detection and tracking to work. However, in case they do exist and are computationally feasible, their input can be utilized towards reducing the number of nodes involved in the grid, further reducing network and processing costs. We are also working on ways to better organize the *Phenomenon-Set* to enable updating and lookup of specific clouds and their characteristics if possible, in constant time. We are also actively searching for real-life datasets from different application domains for further validation and fine-tuning of our approach.

LIST OF REFERENCES

[1] K. Aberer, M. Hauswirth, and A. Salehi. A middleware for fast and flexible sensor network Deployment. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1199-1202, Seoul, Korea, 2006.

[2] M. Ali, W. Aref, R. Bose, A. Elmagarmid, A. Helal, I. Kamel, and M. Mokbel. NILE-PDT: A phenomenon detection and tracking framework for data stream management systems. In *Proceedings of the 31st International Conference on Very Large Data Bases*, pages 1295-1298, Trondheim, Norway, 2005.

[3] M. Ali, M. Mokbel, W. Aref, and I. Kamel. Detection and tracking of discrete phenomena in sensor-network databases. In *Proceedings of the 17th International Conference on Scientific and Statistical Database Management*, pages 163-172, Santa Barbara, USA, 2005.

[4] S. Bhattacharya, N. Atay, G. Alankus, C. Lu, B. Bayazit, and G.-C. Roman. Roadmap query for sensor network assisted navigation in dynamic environments. In *Proceedings of the International Conference on Distributed Computing in Sensor Systems*, pages 17-36, San Francisco, USA, 2006.

[5] R. Bose and A. Helal. Selectivity-based query plan generation in service-oriented sensor networks. In *Technical Report*. Department of Computer and Information Science and Engineering, E305 CSE Building, University of Florida, Gainesville, USA, 2007. http://www.cise.ufl.edu/~rbose/TSH-TR.pdf.

[6] R. Bose, J. King, H. El-Zabadani, S. Pickles, and A. Helal. Building plug-and-play smart homes using the Atlas Platform. In *Proceedings of the 4th International Conference on Smart Homes and Health Telematics*, pages 265-272, Belfast, Northern Ireland, 2006.

[7] K. Chintalapudi and R. Govindan. Localized edge detection in sensor fields. In *Proceedings of the 1st IEEE Workshop on Sensor Network Protocols and Applications*, pages 59-70, Anchorage, USA, 2003.

[8] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-driven data acquisition in sensor networks. In *Proceedings of the 30th International Conference on Very Large Data Bases*, pages 588-599, Toronto, Canada, 2004.

[9] S. de Deugd, R. Carroll, K. Kelly, B. Millett, and J. Ricker. SODA: Service-oriented device architecture. *IEEE Pervasive Computing*, 5(3): 94-96, 2006.

[10] J. Garcia, A. Robertson, J. Ortega, and R. Johansson. Sensor fusion for compliant robot motion control. *IEEE Transactions on Robotics*, 24(2): 430-441, 2008.

[11] G. Hartl and B. Li. Infer: Bayesian inference approach towards energy-efficient data collection in dense sensor networks. In *Proceedings of the 25th IEEE International Conference on Distributed Computing Systems*, pages 371-380, Columbus, USA, 2005.

[12] J. Heidemann, F. Silva, and D. Estrin. Matching data dissemination algorithms to application requirements. In *Proceedings of the 1st International Conference on Embedded Networked Sensor Systems*, pages 218-229, Los Angeles, USA, 2003.

[13] A. Helal, W. Mann, H. El-Zabadani, J. King, Y. Kaddoura, and E. Jansen. The Gator Tech Smart House: A programmable pervasive space. *IEEE Computer*, 38(3): 50-60, 2005.

[14] R. Jiang and B. Chen. Fusion of censored decisions in wireless sensor networks. *IEEE Transactions on Wireless Communications*, 4(6):2668-2683, 2005.

[15] P. Juang, H. Oki, Y. Wang, M. Martonosi, L. Peh, and D. Rubenstein. Energy-efficient computing for wildlife tracking: Design tradeoffs and early experiences with ZebraNet. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, pages 96-107, San Jose, USA, 2002.

[16] S. Kabadayi, A. Pridgen, and C. Julien. Virtual sensors: Abstracting data from physical sensors. In *Proceedings of the IEEE International Symposium on a World of Wireless, Mobile and Multimedia Networks*, pages 587-592, Niagara Falls, USA, 2006.

[17] Y. Kaddourah, J. King, and A. Helal. Cost-precision tradeoffs in unencumbered floor-based indoor location tracking. In *Proceedings of the 3rd International Conference on Smart Homes and Health Telematics*, pages 75-82, Sherbrooke, Canada, 2005.

[18] J. King, R. Bose, S. Pickles, A. Helal, and H. Yang. Atlas – A service-oriented sensor platform. In *Proceedings of the 1st IEEE International Workshop on Practical Issues in Building Sensor Network Applications*, pages 630-638, Tampa, USA, 2006.

[19] C. Lee, D. Nordstedt, and A. Helal. Enabling smart spaces with OSGi. *IEEE Pervasive Computing*, 2(3): 89-94, 2003.

[20] F. Lewis. Wireless sensor networks. *Smart Environments: Technologies, Protocols and Applications*, pages 13-46, John Wiley, 2004.

[21] X. Liu, Q. Huang, and Y. Zhang. Combs, needles and haystacks: Balancing push and pull for discovery in large-scale sensor networks. In *Proceedings of the 2nd International Conference on Embedded Networked Sensor Systems*, pages 122-133, Baltimore, USA, 2004.

[22] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th Annual Symposium on Operating Systems Design and Implementation*, pages 131-146, Boston, USA, 2002.

[23] S.Madden, M. Franklin, J. Hellerstein, and W. Hong. TinyDB: An acquisitional query processing system for sensor networks. *ACM Transactions on Database Systems*, 30(1): 122-173, 2005.

[24] N. Malpani, J. Welch, and N. Vaidya. Leader election algorithms for mobile ad hoc networks. In *Proceedings of the 4th International Workshop on Discrete Algorithms and Methods for Mobile Computing and Communications*, pages 96-103, Boston, USA, 2000.

[25] D. McErlean and S. Narayanan. Distributed detection and tracking in sensor networks. In *Proceedings of the 36th Asilomar Conference on Signals, Systems and Computers*, pages 1174-1178, Asilomar, USA, 2002.

[26] R. Olfati-Saber and J. Shamma. Consensus filters for sensor networks and distributed sensor fusion. In *Proceedings of the 44th IEEE Conference on Decision and Control and the European Control Conference*, pages 6698-6703, Seville, Spain, 2005.

[27] A. Omotayo, M. Hammad, and K. Barker. Efficient data harvesting for tracing phenomena in sensor networks. In *Proceedings of the 18th International Conference on Scientific and Statistical Database Management*, pages 59-70, Vienna, Austria, 2006.

[28] S. Shenker, S. Ratnaswamy, B. Karp, R. Govindan, and D. Estrin. Data-centric storage in sensornets. *ACM SIGCOMM Computer Communication Review*, 33(1): 137-142, 2003.

[29] A. Silberstein. Push and pull in sensor network query processing. In *Proceedings of the Southeast Workshop on Data and Information Management*, Raleigh, USA, 2006.

[30] A. Silberstein, A. Gelfand, K. Munagala, G. Puggioni, and J. Yang. Making sense of suppressions and failures in sensor data: A Bayesian approach. In *Proceedings of the 33rd International Conference on Very Large Data Bases*, pages 842-853, Vienna, Austria, 2007.

[31] C. Town and Z. Zhu. Sensor fusion and environmental modeling for multimodal sentient computing. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, pages 1-2, Minneapolis, USA, 2007.

[32] N. Trigoni, Y. Yao, A. Demers, J. Gehrke, and R. Rajaraman. Hybrid push-pull query processing for sensor networks. *GI Jahrestagung*, (2): 370-374, 2004.

[33] S. Vasudevan, B. DeCleene, N. Immerman, J. Kurose and D. Towsley. Leader election algorithms for wireless ad hoc networks. In *Proceedings of the 3rd DARPA Information Survivability Conference and Exposition*, pages 261-272, Washington DC, USA, 2003.

[34] A. Woo and D. Culler. A transmission control scheme for media access in sensor networks. In *Proceedings of the 7th Annual International Conference on Mobile Computing and Networking*, pages 221-235, Rome, Italy, 2001.

BIOGRAPHICAL SKETCH

Raja Bose was born in Darjeeling in 1981 and raised in New Delhi, India. He has a Bachelor of Science degree (with Honors) in statistics from Delhi University, India and two Master of Science degrees from the University of Florida: one in applied mathematics and the other in computer engineering. Since 2005, he has been conducting research under the guidance of Dr. Abdelsalam (Sumi) Helal, in the area of service-oriented sensor networks and smart spaces, at the Mobile and Pervasive Computing Laboratory of the University of Florida. He was awarded the Doctorate in Computer Engineering in May 2009 and joined Nokia Research Center Palo Alto in January 2009, as a member of research staff specializing in ubiquitous device interoperability.