The Dissertation Committee for Sanem Kabadayi
certifies that this is the approved version of the following dissertation:

Enabling Programmable Ubiquitous Computing Environments: The DAIS Middleware

Committee:

Christine Julien, Supervisor

Tony Ambler

William Bard

William O'Brien

Dewayne Perry

Enabling Programmable Ubiquitous Computing Environments: The DAIS Middleware

by

Sanem Kabadayi, B.S.E.E; B.S.Phy.; M.S.E.E.C.S.

Dissertation

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

Doctor of Philosophy

The University of Texas at Austin

May 2008

To the memory of Mustafa Kemal Atatürk

and

to my family

Acknowledgments

I would like to thank my dissertation supervisor, Christine Julien, for her guidance and valuable suggestions. She is living proof that wisdom and maturity are not necessarily linked to age. I have learned a lot from her creativity and dedication. I will always appreciate her treating us (all of her students) as colleagues. I am grateful that she has given me a chance to work with her and will always be proud to have been her student. Without her, this dissertation would never have been possible. I am grateful to William Bard for always supporting me and his constancy of character in all the eight years that I have known him. Without Tony Ambler's help, this dissertation would never have been possible, either. He truly cares about each student in the department not only because he is the department chair, but because they are fellow human beings, and he goes above and beyond the call of duty to make sure they are treated fairly. I would also like to thank William O'Brien and Dewayne Perry for always speaking very highly of my work and being very encouraging. I am grateful to all these members of my dissertation committee for their valuable feedback on this work.

In the department, I would like to thank Diana Perez for being such a kind and caring person, Melanie Gulick for her smiling face, encouragement, and help, and Charlotte Harris for all her help. I thank all of the Mobile and Pervasive Computing (MPC) Group members for their support and encouragement. It has truly been a pleasure working with Dr. O'Brien and his students, Thuy Nguyen and Xiaowei Luo, on the application examples from the construction site domain and Randy Hennig from the Faculty Innovation Center for the collaboration on the CI-TEAM demo. I also appreciate all the help Loren Rittle (Motorola Labs) has given me in understanding Melete.

I could not have completed this Ph.D. without the support of the great friends I am truly lucky to have. I cannot thank Huihui Wang and Huijie Xu enough for being like my family away from home and sharing their Ph.D. experiences with me; I am proud to be their *xiao-mei* ("little sister"). I also appreciate all the support I have received from Liping Feng, Shiju Wang, and Xiaohong Li. I am grateful to Canay Tulunay-Riordan for sharing my undergraduate experience, as well as being in Austin the first few years of my Ph.D., and constantly supporting me. I am truly grateful to Hande Eren for being a confidante, meeting me for a coffee whenever I needed to talk to a close friend, and always having nice things to say. I cannot thank Claudine D'Annunzio enough for helping me get through the final stages of writing this dissertation by studying with me and her encouraging "*Belle, bonne et capable!*" motto. I am thankful to Ania Kacewicz for her encouraging words and support. I am grateful to Ariton Xhafa for getting me started on being a referee for conference papers; this experience has enabled me to put my own papers in a different perspective. I am also grateful to Melis Ekinci, Burcu Hacıbaşıoğlu-Geniş, Ece Saygun, and Melek Seyrekoğlu for having infinite trust in my capabilities and their continuous support.

I am forever grateful to my family for their unconditional love and support. They have taught me so much and always set the finest examples of professionalism. In addition to my parents, my brother has also given me wonderful advice whenever I needed it through my Ph.D. experience. They have always been on my side and always given me things to look forward to. I am also grateful to Istanbul for always inspiring me, for it is impossible to create anything new without a muse.

Veni, vidi, ... <u>vixi</u>!

Sanem Kabadayi

The University of Texas at Austin

May 2008

Enabling Programmable Ubiquitous Computing Environments: The DAIS Middleware

Publication No. ____________________

Sanem Kabadayi, Ph.D.
The University of Texas at Austin, 2008

Supervisor: Christine Julien

Emerging ubiquitous computing scenarios involve client applications that dynamically collect information directly from the local environment by leveraging sensor network nodes opportunistically and unpredictably. Such scenarios deviate from existing deployments of sensor networks which are often highly application-specific and generally funnel information to a central collection service for a single purpose. A significant barrier to the widespread development of such flexible sensor network applications lies in the increased complexity of the programming task when compared to existing distributed or even mobile situations. Ubiquitous computing nodes are severely resource-constrained, in terms of both computational capabilities and battery power, and therefore the application development task must inherently consider low-level design concerns, such as reducing power consumption, minimizing communication in the network to extend the network's lifetime, and handling the variability of devices' capabilities and constraints. This complexity, coupled with the increasing demand for applications, highlights the need for programming platforms (i.e., middleware) that simplify application development.

This dissertation reports on the DAIS (Declarative Applications in Immersive Sensor networks) middleware platform that enables the development of adaptive ubiquitous computing applications. Our approach focuses on minimizing communication and coordination to best ensure the network's lifetime. DAIS attempts to localize data collection and sensor interaction to only the regions of the network required for the applications' immediate data needs. At the programming interface level, this requires exposing some aspects of the physical world to the developer, and we accomplish this through novel programming abstractions that enable on demand access to dynamic data sources. We develop a pair of intuitive grouping abstractions, the scene (which enables local interactions) and the virtual sensor (which enables automatic abstraction of heterogeneous data), to define a coordination model that supports interactions in ubiquitous computing. We combine these abstractions with an expressive programming interface to create the complete DAIS middleware.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Ubiquitous (also called pervasive) computing environments entail a multitude of ubiquitous, embedded devices that provide information about an environment. These devices have limited computational and communication capabilities and generally provide highly-specialized functionality. Ubiquitous computing applications themselves commonly run on "client" devices (e.g., handheld or laptop computers) that users carry through the environment. As developers construct applications for these environments, their code must be able to adjust to different physical devices and explicitly handle discovery of and communication with embedded resources.

To date, much application development for ubiquitous computing environments has been limited to academic circles. Many new concerns arise in comparison to existing distributed or mobile computing scenarios, increasing the complexity of the programming task. This complexity, coupled with the increasing user demand for flexible and expressive applications, highlights the need for a changed perspective on the computing environment and the encapsulation of this perspective in programming platforms (i.e., *middleware*) that simplify application development.

Ubiquitous computing applications from domains such as intelligent construction sites [20], aware homes [35], and pervasive office environments [56] involve users immersed in the network accessing locally-sensed information on demand. Such application scenarios motivate a view of the ubiquitous computing environment that is device-agnostic. That is, applications, in general, do not care about devices embedded in the environment but instead about information and resources available locally. Logically, the ubiquitous environment appears as a world of embedded information available for immersed applications. To support this view, it is essential to allow developers to distance themselves from explicit knowledge of devices and communication capabilities of the underlying ubiquitous computing network and instead focus on dynamically changing, locally available information.

In ubiquitous computing applications that rely on immersive sensor networks, users with client devices need to interact directly with devices (or sensors) embedded in their environments. This allows the client applications to operate over information collected directly from the local area (as shown in Figure 1.1(b)). This is in contrast to existing sensor network deployments in which sensor networks are commonly accessed through a central collection point (as shown in Figure 1.1(a)). The protocols available to support communication and coordination on lightweight, resource-constrained sensors are tailored to application situations like those in Figure 1.1(a). This changing style of interaction is a direct motivation for a reexploration of protocol and coordination issues in immersive computing environments. More directly, immersive sensor networks support ubiquitous computing applications (as in Figure 1.1(b)), not remote distributed sensing (as in Figure 1.1(a)). In such ubiquitous computing environments, we differentiate client devices (those on which ubiquitous computing applications run) from sensors (devices embedded in the environment). The former commonly support users and have increased computational power, while the latter are heavily resource-constrained.

The style of interaction apparent in ubiquitous computing and depicted in Figure 1.1(b) differs from common uses of sensor networks, introducing several

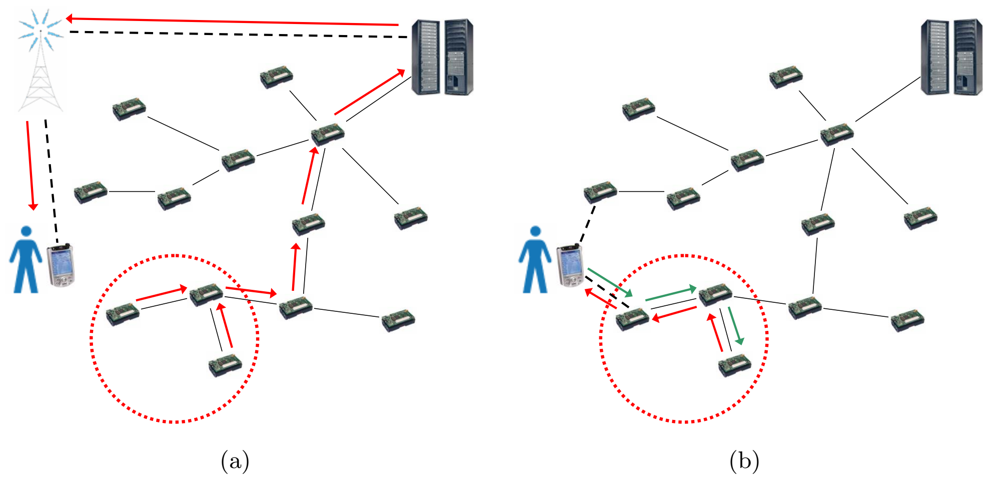


Figure 1.1: Comparison of (a) existing operational environments and (b) immersive sensor networks for ubiquitous computing

unique challenges and heightening existing ones:

- *Locality of interactions*: An application interacts directly with local sensor nodes. Since the interactions between the client device and the sensing nodes no longer have to go through a distant central collection point or a gateway, and redundant broadcasts from several sensors can be combined in the network, the communication overhead and latency can be minimized. However, it can also be cumbersome with respect to enabling the application to precisely specify the area from which it collects information.

- *Mobility-induced dynamics*: While the sensor nodes are likely stationary (as in many other deployments), the application interacting with them runs on a device carried by a mobile user. Therefore, the device's connections to particular sensors and the area from which the application desires to draw information are subject to constant change.

- *Unpredictability of coordination*: To support future ubiquitous computing environments, it is essential that the network be general-purpose. As such, few *a* *priori* assumptions can be made about the needs or intentions of applications, requiring the network to adapt to unexpected and changing situations.

- *Complexity of programming*: In addition, the desire to provide end-user applications (as opposed to more database-oriented data collection) increases the demand for applications and the number of programmers that will need to construct them.

The confluence of these challenges necessitates a flexible yet expressive programming environment that enables ubiquitous computing application development while paying careful attention to resource constraints of embedded devices. To meet the needs of the intended applications, a middleware is required that allows an application running on a user's device to seamlessly connect to the resources embedded in the local region, removing the requirement that physically distant resources participate in a query's resolution.

To address these issues, we have created a middleware platform DAIS [1] (Declarative Applications in Immersive Sensor networks) that provides programming abstractions tailored to ubiquitous applications. This is not a middleware for sensor networks in the sense that it runs strictly on the sensors. Instead, it allows developers to create applications that run on *client devices* (e.g., laptops or PDAs) that interact directly with embedded networks. Our approach minimizes the complexity of communication using a pair of intuitive abstractions: the *scene*, which enables local interactions, and the *virtual sensor*, which enables automatic abstraction of heterogeneous data.

The specific novel contributions that of this work can be categorized into model-level and implementation-level contributions. A coordination model defined by the two key abstractions is created to support interaction between client devices and a ubiquitous computing environment. The model is realized in a middleware

[1]DAIS (dā′ĭs): from the middle English word meaning "raised platform."

implementation whose performance and usefulness is evaluated through simulation and prototyping.

With respect to resolving the challenges enumerated above, the contributions of this work are the following:

1. Basic protocols for scene construction and maintenance and their evaluation.
2. Formalization of a programming model for the specification of scenes by novice developers.
3. Formal definition virtual sensors and their programming model.
4. Intelligent algorithms for deploying and dynamically redeploying virtual sensors.
5. A coherent middleware that encapsulates both the virtual sensor and the scene and provide an integrated high-level programming interface.

We focus on supporting applications in which client devices (e.g., laptops or PDAs) interact directly with embedded sensor networks. While this style of interaction is common in many application domains, we will refer to applications from the intelligent construction site domain and the first responder domain, both of which provide a unique and heterogeneous mix of embedded and mobile devices. An intelligent construction site consists of users with mobile devices distributed over the site and sensors embedded in equipment. Building applications for intelligent construction sites presents substantial challenges to ubiquitous computing. In the first responder application domain, the embedded devices include fixed sensors in buildings and environments that are present regardless of crises and ad hoc deployments of sensors that responders may distribute when they arrive. Mobile devices include those moving within vehicles, carried by responders, and even autonomous robots that may perform exploration and reconnaissance.

Parts of this dissertation have been published in conferences and journals [27, 28, 30, 31, 34].

The remainder of this dissertation is organized as follows. Chapter 2 undertakes the explanation of the development and implementation of the scene abstraction. Chapter 3 presents the virtual sensor model and two example example applications. Chapter 4 explores mechanisms for remote deployment of virtual sensors. Chapter 5 presents the resulting middleware, DAIS, detailing the model and its implementation. Chapter 6 summarizes the work, provides a brief overview of future work, and concludes this dissertation.

Chapter 2

Communication Underpinnings: The Scene Abstraction

In ubiquitous computing applications, users move through instrumented environments and desire on-demand access to locally gathered information. The set of data sources near a user changes based on the user's movement. Furthermore, if the network is well-connected, the user's device will be able to reach vast amounts of raw information that must be filtered to be usable. The application must be able to limit the scope of its interactions to include only the data that matches its needs. We encapsulate an application's operating environment (i.e., the sensors and devices with which it interacts) in an abstraction called a scene. This abstraction allows local, multihop neighborhoods of heterogeneous devices surrounding an application, supports mobility of the user by dynamically updating the scene's participants in response to environmental changes, and minimizes how much the application developer must know about the implementation of the underlying network. In this chapter, we first overview other approaches to defining similar relative neighborhoods. We then briefly overview our work on the scene abstraction, which has demonstrated the feasibility of providing changing local neighborhoods to application users. We conclude this chapter by examining the research contributions with respect to the scene abstraction in detail.

2.1 Lightweight Grouping Mechanisms

Previous work has investigated group and neighborhood abstractions in both mobile ad hoc networks and sensor networks. In mobile ad hoc networks, the network abstractions model [52] allows applications to provide metrics over network paths. Nodes to which there exists a path satisfying the metric are included in the specifier's *network context*, while those outside are excluded. This approach is overly expressive for ubiquitous computing applications, making it difficult to specify simple metrics. In addition, the protocol does not function on resource-constrained nodes. SpatialViews [49] abstracts properties of mobile ad hoc networks to enable applications to be developed in terms of virtual networks defined by characteristics of the underlying physical network. SpatialViews focuses on the distributed computations that occur in the defined virtual networks, while we focus instead on defining such virtual networks and on the underpinnings of communication that hold those groups together. Collaboration groups, defined as part of *state-centric programming* [40], abstract common patterns in application-specific communication and resource allocation. However, the focus of collaboration groups is defining a programming model, not the communication model necessary to efficiently support such abstractions.

In sensor networks, several approaches provide neighborhood or regional abstractions to scope applications' interactions. Hood [59] allows sensor nodes to define neighborhoods of coordination around themselves based on network properties. The implementation only allows neighborhoods that extend a single hop, while we posit that multiple-hop neighborhoods are necessary for expressive ubiquitous computing interactions. In addition, Hood does not enable dynamic updates to a neighborhood's participants, making it unsuitable for supporting moving users and changing environments. Abstract Regions [58] define regions of coordination and couple the abstraction with programming constructs that allow applications to issue operations over the regions. Likewise, *logical neighborhoods* [46] provide a communication infrastructure that logically groups similar nodes. This logical grouping of nodes is based on a notion of proximity defined by the application, and it is not necessarily a neighborhood based on underlying physical properties. These approaches do not directly consider the dynamics of mobility, and they require proactive behavior by all sensors all the time.

While the above approaches do not address dynamics, a few constructs have begun to do so. Mobicast [23] defines a message dissemination algorithm for pushing messages to nodes that fall in a region in front of a moving target. As the target moves through a field of sensors, the protocol dynamically changes the field of receivers in response to the movement. MobiQuery [41] also supports spatiotemporal queries and allows a query area to respond to a user's announced motion profile. These approaches require nodes to have a fine-grained knowledge of their physical locations, which is not a reasonable assumption for future ubiquitous computing networks in which the sensor nodes and their deployments must be inexpensive and require minimal setup and administration. EnviroTrack [1] is tailored to providing object tracking by a dynamic group of sensor nodes. This system focuses on identifying and labeling tracked objects so that they can be addressed using more traditional communication. EnviroTrack has been extended by EnviroSuite [42] and its associated communication protocol [4]. Communication again relies on each node knowing its exact physical location. This is an acceptable assumption in these systems, given that the goal is often to know the physical location of some tracked object. In ubiquitous computing, however, relative locations often suffice to support applications' group formation. Supporting local interactions in ubiquitous computing environments requires specifying only the relation the nodes must have to the user and is not concerned with the exact locations. For example, on an intelligent construction site, a supervisor would be interested in the cranes on the site that are a certain number of hops away from him, but not necessarily the cranes' GPS coordinates. Obtaining GPS coordinates would unnecessarily increase the computational tasks and cost of a ubiquitous computing application.

Creating a communication paradigm that supports ubiquitous computing applications requires a protocol that provides a facility for enabling direct, opportunistic interactions among heterogeneous devices and sensors. This protocol needs to support the mobility of this regional abstraction without relying on knowledge of absolute locations.

2.2 Scenes: Abstractions of Local Data

In our model, an application's operating environment (i.e., the sensors with which it interacts) is encapsulated in a novel abstraction we have developed, called a *scene* [28]. Applications define scenes according to their needs, and each scene constrains which particular sensors may influence the application. The constraints may be on properties of hosts (e.g., battery life), of network links (e.g., bandwidth), and of data (e.g., type). These types of constraints offer generality and flexibility and provide a higher level of abstraction to the application developer.

2.2.1 Defining Scenes

The declarative specification defining a scene allows an application programmer to flexibly describe the type of scene he wants to create, and multiple constraints can be used to define a single scene. The programmer only needs to specify three parameters to define a single scene constraint:

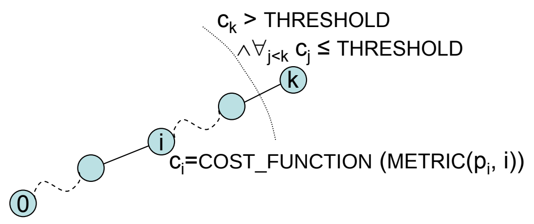

c_k > THRESHOLD
$\wedge \forall_{j<k}\, c_j \leq$ THRESHOLD
k
i
0
c_i=COST_FUNCTION (METRIC(p_i, i))

Figure 2.1: Distributed scene computation

- *Metric*: A property of the network or environment that defines the cost of a connection (i.e., a property of hosts, links, or data).
- *Path cost function*: A function (such as sum, average, minimum, maximum) that operates on a network path to calculate the cost of the path.
- *Threshold*: The value a path's cost must satisfy for that sensor to be a member of the scene.

Thus, a scene, S, is specified by one or more constraints, C_1, C_2, C_3, ... C_n:

$$C_1 = \langle M_1,\ F_1,\ T_1\rangle,\ C_2 = \langle M_2,\ F_2,\ T_2\rangle,\ C_3 = \langle M_3,\ F_3,\ T_3\rangle,\ \dots\ ,\ C_n = \langle M_n,\ F_n,\ T_n\rangle$$

where M denotes a metric, F denotes a path cost function, and T denotes a threshold.

Figure 2.1 demonstrates the relationships between these components. This figure is a simplification that shows only a one-constraint scene and a single network path. The cost to a particular node in the path (e.g., node i) is calculated by applying the scene's path cost function to the metric. The latter can combine information about the path so far (p_i) and information about this node. Nodes along a path continue to be included in the scene until the path reaches a node whose cost (e.g., c_k) is greater than the scene's threshold. This functionality is implemented in a dynamic distributed algorithm that can calculate (and dynamically recalculate) scene membership. The application's messages carry with them the metric, path cost function, and threshold, which suffice to enable each node to independently determine whether it is a member of the scene. Each node along a network path determines whether it lies within the scene, and if so, forwards the message. It is possible for a node to qualify to be within the scene based on multiple paths, and the one with the least cost is chosen. The selected network paths correspond to branches of a routing tree that is set up as part of the distributed calculation of the scene. When a certain data source needs to relay a reply back to the user, the reverse of the path on the routing tree the message took to get to that node can be used. If a node receives a scene message that it has already processed, and the new metric value is not shorter, the new message is dropped. If the new message carries a shorter metric, then the node forwards the information again because it may enable new nodes to be included in the scene.

This scene definition can be formalized in the following way:

Given a client node α, a metric M, and a positive threshold T, find the set of all hosts S_α such that all hosts in S_α are reachable from α and, for all hosts β in S_α, the cost of applying the metric on some path from α to β is less than T. Specifically:

$$S_\alpha = \langle set\ \beta\colon\ M(\alpha, \beta) < T :: \beta \rangle$$[1]

We note that the above formalization is for a scene that uses only one metric; if the scene is specified by multiple metrics, the formalization must require that the nodes in the returned set satisfy all of the metric/threshold pairs.

The scene concept conveys a notion of locality, and each application decides how "local" its interactions need to be. A construction site supervisor coordinating a team spread throughout the site may want to have an aggregate view of the

[1] In the three-part notation: ⟨`op` *quantified_variables* : *range* :: *expression*⟩, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which `op` is applied, yielding the value of the three-part expression. If no instantiation of the variables satisfies *range*, then the value of the three-part expression is the identity element for `op`, e.g., *true* if `op` is $\forall$, or $\emptyset$ if `op` is *set*.

dangerous gas cloud conditions over the entire site. On the other hand, a particular worker may want a scene that contains readings only from nearby sensors or sensors within his path of movement. The scene for the supervisor would be "all gas sensors within the site boundaries," while the scene for the worker might be "all gas sensors within 5m." As a worker moves through the site, the scene *specification* stays the same, but the data sources belonging to the scene may change.

To maintain the scene for continuous queries, each member sends periodic beacons advertising its current value for the metric. Each node also monitors beacons from its parent in the routing tree, whose identity is provided as previous hop information in the original scene message. If a node has not heard from its parent for a specified amount of time (the beacon interval), it disqualifies itself from the scene. This corresponds to the node falling outside of the span of the scene due to client mobility or other dynamics. In addition, if the client's motion necessitates a new node to suddenly become a member of the scene, this new node becomes aware of this condition through the beacon it receives from a current scene member.

Table 2.1 shows examples of how scenes may be specified. These examples include restricting the scene by the maximum number of hops allowed, the minimum allowable battery power on each participating node, or the maximum physical distance. As one example, SCENE_HOP_COUNT effectively assigns a value of one to each network link. Therefore, using the built-in SCENE_SUM path cost function, the application can build a hop count scene that sums the number of hops a message takes and only includes nodes that are within the number of hops as specified by the threshold. The scene can be further restricted using latency as a second constraint.

Our contributions are the novel model of coordination and prototype protocols for supporting that model. In the remainder of this section, we will first discuss results of a feasibility study of this basic implementation and then the fundamental research issues that we plan to undertake as part of this dissertation research.

Table 2.1: Example scene definitions

	Hop Count Scene
Metric	SCENE_HOP_COUNT
Aggregator	SCENE_SUM
Metric Value	number of hops traversed
Threshold	maximum number of hops
	Battery Power Scene
Metric	SCENE_BATTERY_POWER
Aggregator	SCENE_MIN
Metric Value	minimum battery power
Threshold	minimum allowable battery power
	Distance Scene
Metric	SCENE_DISTANCE
Aggregator	SCENE_DFORMULA
Metric Value	location of source
Threshold	maximum physical distance

2.2.2 A Programming Interface for Scenes

To present the scene to the developer, we build a simple API that includes general-purpose metrics (e.g., hop count, distance, etc.) and provides a straightforward mechanism for inserting new metrics. Applications specify scenes through a Java programming interface, and these specifications are translated into low-level sensor code. Fig. 2.2 shows the Java API, which relies on a `Query` that the application provides when interacting with a `Scene`. This `Query` should be delivered to every member of the `Scene`. The `ResultListener` interface used in the `Scene` API allows nodes who are members of the scene to return responses to the client device. The `Scene` API intentionally does not restrict what kind of application-level communication these query and reply interactions can encode; this depends on the particular application layer running on both the client device and the sensor. The scene abstraction simply provides expressive connectivity among these components. In Section 2.4, we will explore a simple application layer for a first responder application example.

From the application's perspective, a scene is a dynamic data structure containing a set of qualified sensors, which are determined by a list of con-

```
class Scene{
   public Scene(Constraints[] c);
   public void send(Query q, ResultListener rl);
   public void maintain(Query q, ResultListener rl, int frequency);
}
```

Figure 2.2: The API for the `Scene` class

straints, `Constraints[]`, and accessed through the latter two methods: `send()` and `maintain()`. The `send()` method poses a one-time query to scene members to which each recipient sends at most one reply. This is similar to a multicast, but the significant difference is that the receivers are *dynamically* determined by the parameters defining the scene. The `maintain()` method sends a persistent query to the scene, implicitly requesting that the scene structure be maintained, even as the participants change. The `frequency` parameter in the `maintain` method indicates how often the application expects responses.

The `Scene` API is intentionally simple. It focuses on providing access to the scene constructs and not on incorporating client functionality into the scene communication components. We are motivated to keep the API as slim as possible to ease the implementation on resource-constrained devices. By limiting the functionality available to applications, we more closely match the capabilities of this underlying constrained hardware. While only the one-time query behavior is essential, the `maintain` operation enables a more efficient implementation of persistent queries. This is especially important in resource-constrained networks, where minimizing communication overhead is essential.

2.2.3 Maintaining Scenes

While a scene provides the appearance of a dynamic data structure, the implementation behaves on demand; no proactive behavior occurs. Only when the application uses a scene does the protocol communicate with other local devices, reducing the overall communication overhead. At first glance, this approach may appear to incur an unpredictable latency for the first query posed to a scene. However, queries traverse the same path as the scene construction messages, and the queries themselves carry the scene construction information. Therefore, the on-demand construction incurs no additional latency.

For one-time queries, a scene is created, and the scene information is not stored or updated in any way. However, if the scene is to be used for a persistent query, it needs to be maintained. To maintain the scene for such continuous queries, each member sends periodic beacons advertising its current value for the metric. Each node also monitors beacons from its parent in the routing tree, whose identity is provided as previous hop information in the original scene message. If a node has not heard from its parent for three consecutive beacon intervals, it disqualifies itself from the scene. This corresponds to the node falling outside of the span of the scene due to client mobility or other dynamics. In addition, if the client's motion necessitates a new node to suddenly become a member of the scene, this new node becomes aware of this condition through the beacon it receives from a current scene member.

Pervasive applications expect access to locally available resources. Consider an application in an aware home. An application may connect to resources that it can monitor and control within the room the user occupies. As the user moves around the home, the scope of this control should change to match the user's changing rooms. Therefore, we provide the automatic maintenance described above instead of calculating a static scene when the application initially declares it. This style of maintenance is particularly well-suited to pervasive computing applications, which demand automated context-awareness.

2.2.4 Defining Scenes Based on Physical Characteristics

The metrics used to specify scenes can be divided into two categories: those that define scenes based on properties of *network paths* or the devices on the network paths (e.g., latency or battery power) and those that define scenes based on *physical characteristics* of the environment (e.g., location or temperature).

Using a physical characteristic to calculate network paths is plagued by the *C-shaped network problem* [30]. Consider the network shown in Fig. 2.3. Nodes A and B are within 50m of each other, yet a discovery from A to B must leave the region of radius 50m surrounding A to find B. The only way to guarantee that every device is discovered is to flood the entire network. The network abstractions model [52] directly recognizes this situation and *guarantees* that it calculates a correct region by requiring applications' region definitions to include metrics that strictly increase along a network path. Absolute physical distance is not such a metric, so to fit into this model, it must be combined with another metric that does satisfy the requirement (e.g., hop count). Abstract Regions [58] implicitly addresses this issue by enabling only *geographic filters* on neighborhoods first defined by hop count. Hood [59] avoids this trouble altogether by limiting collection neighborhoods to one-hop regions and arguing that such regions meet the demands of current applications.

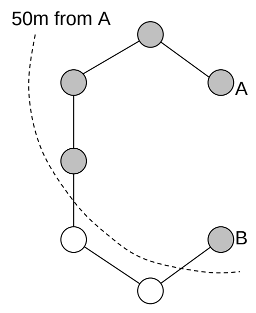
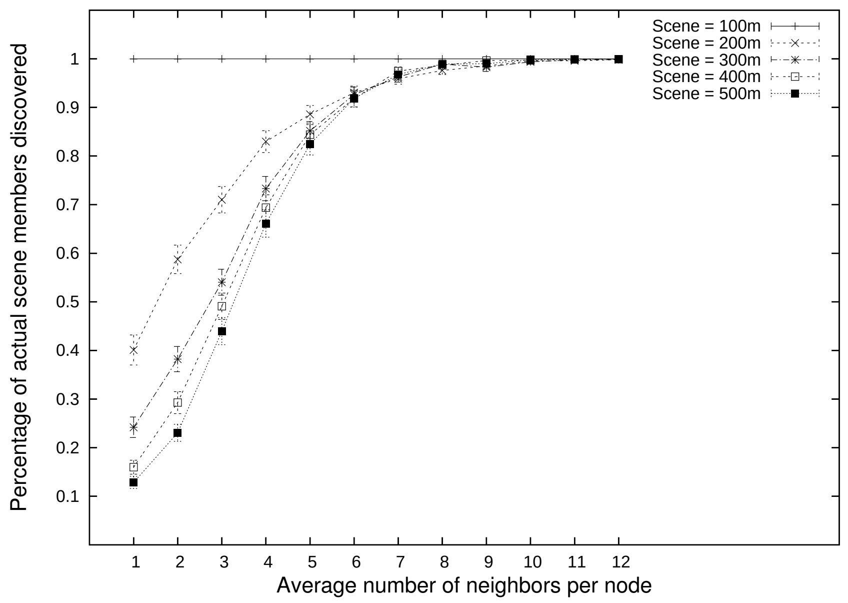


Figure 2.3: A C-network

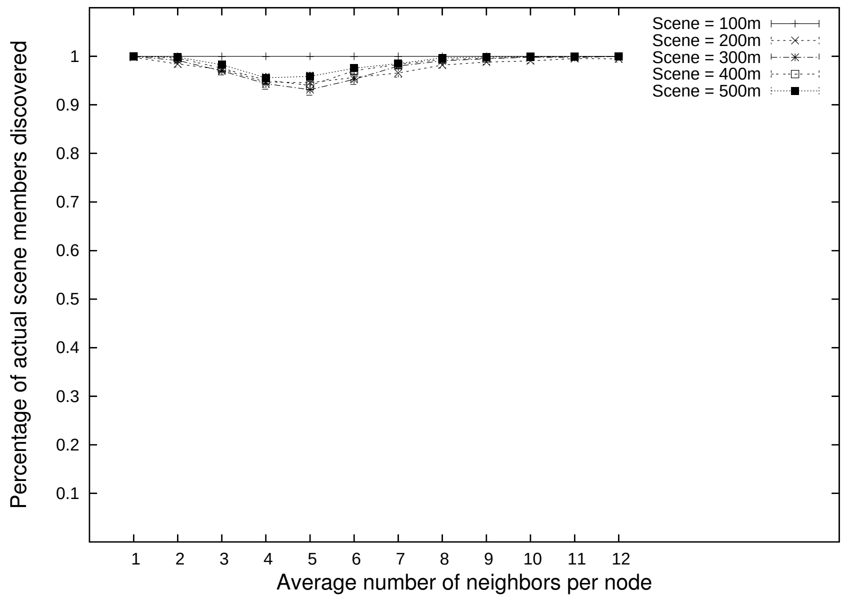
In the types of pervasive computing applications we address, an application may not be in direct communication with the devices with which it needs to interact. In a first responder situation, safety applications may dictate that each user has information about a region larger than a device's communication radius, for example to monitor the presence and movement of fire or gases. For this reason, we focus on building the best multihop neighborhoods possible. For metrics that measure physical characteristics, the question remains as to how to handle the ambiguity separating the natural specification (e.g., "all devices within 50m") and the ability of a protocol to efficiently satisfy that specification (i.e., without flooding the network). Given our experience with the complexity involved in creating region specifications using network abstractions, we favor an approach that does not require strictly increasing metrics. This makes the programming interface simpler, but in the presence of configurations like that shown in Fig. 2.3, our approach may not find some members of the specified scene even though they are transitively connected.

Figs. 2.4(a) and (b) show the results of experiments that demonstrate the ramifications of this design decision. In these experiments, we generated random network topologies in a 1000m^2 space with the following parameters. The number of nodes was randomly selected to be between 20 and 400, and each node was randomly placed. We used a communication radius of 100m, i.e., any two nodes within 100m of each other were considered "neighbors." We constructed scenes based on physical distances ranging from 100m to 500m. A 100m scene includes only nodes within the requester's communication range (i.e., within one-hop). In each graph, the x-axis shows the average number of one-hop neighbors per node. Each point corresponds to 500 samples, and 95% confidence intervals are given. For each sample, one node was randomly selected to request a scene of the specified size.

Fig. 2.4(a) shows the percentage of *actual* scene members discovered by our protocol. This includes every node within the specified physical distance radius, even nodes to which no network connectivity exists. At low network density, the quality of the scene construction was poor, especially as the physical size of the scene increased. This is because the network was so sparsely connected that it was unlikely that nodes were able communicate, especially when they desired to find other nodes at large distances. However, with increasing density, our protocol found more than 90% of the actual scene members.



(a)



(b)

Figure 2.4: Accuracy of location-based scene calculations

Fig. 2.4(b) demonstrates even stronger motivation for our best-effort approach for scenes based on physical characteristics. This graph limits the error expressed to only those scene members that were not discovered but were connected by a finite number of network hops (i.e., those nodes reachable by flooding). The percentage of scene members that our method *did not* discover is never more than 10% and is usually close to 0. The valley corresponds to cases when the network was largely connected but connections were sparse. In these situations, roundabout paths may exist when more direct routes do not. To the left of the valley, the network was largely disconnected, so we do not miss many connected scene members; to the right, the network was much more connected, and the direct approach is quite successful.

The results demonstrate that our approach tends to find the vast majority of the scene members under reasonable conditions. Therefore, we favor natural scene specifications over complete accuracy of scene membership.

2.3 Realizing Scenes on Resource-Constrained Sensors

The model for both construction and maintenance of scenes described in the previous section is tailored to the requirements of pervasive computing environments. In creating applications for client devices, developers can leverage the Java interface from Fig. 2.2, which allows them to use the scene communication abstraction to interface with sensing devices. Software on these sensing devices must also support the scene abstraction. In this section, we describe this implementation, showing how the code is structured to support dynamic, opportunistic communication.

2.3.1 A Structured Implementation Strategy

Our implementation uses the *strategy pattern* [15], a software design pattern in which algorithms (such as strategies for scene construction and maintenance) can be chosen at runtime depending on system conditions. The strategy pattern provides a means to define a family of algorithms, encapsulate each one, and make them interchangeable. Such an approach allows the algorithms to vary independently from the clients that use them. In the scene, the clients that employ the strategies are the queries, and the different strategies are `SceneStrategy` algorithms. Fig. 2.5 shows the resulting architecture. We decouple the scene construction from the code that implements it so that we can vary message dissemination without modifying application-level query processing (and vice versa).

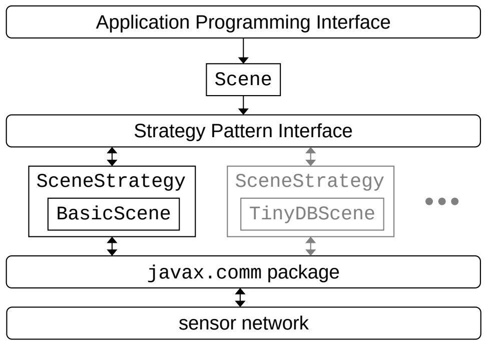


Figure 2.5: Simplified software architecture

The remainder of this section describes one implementation of the `SceneStrategy`, the `BasicScene`, which provides a prototype of the protocol's functionality. Other communication styles can be swapped in for the `BasicScene` (for example one built around TinyDB [44] or directed diffusion [25]). By defining the `SceneStrategy` interface, we enable developers who are experts in existing communication approaches to create simple plug-ins that use different query communication protocols and yet still take advantage of the scene abstraction and its simplified programming interface.

2.3.2 A Basic Instantiation

While the scene abstraction is independent of the particular hardware used to support it, in our initial implementation, these software components have been developed for Crossbow Mica2 motes [12] and are written for TinyOS [22] in the nesC language [16]. Our nesC implementation of the scene abstraction (along with other project information) is available at http://mpc.ece.utexas.edu/scenes/index.html. In nesC, an application consists of *modules* wired together via shared interfaces to form *configurations*. Fig. 2.6 depicts the components of the scene configuration and the interfaces they share.

This implementation functions as a routing component on each node, receiving each incoming message and processing it as our protocol dictates. In this picture, we show components as rounded rectangles and interfaces as arrows connecting components. A component *provides* an interface if the corresponding arrow points toward it and *uses* an interface if the corresponding arrow points away it. If a component provides an interface, it must implement all of the *commands* specified by the interface, and if a component uses an interface, it can call any commands declared in the interface and must handle all *events* generated by the interface.

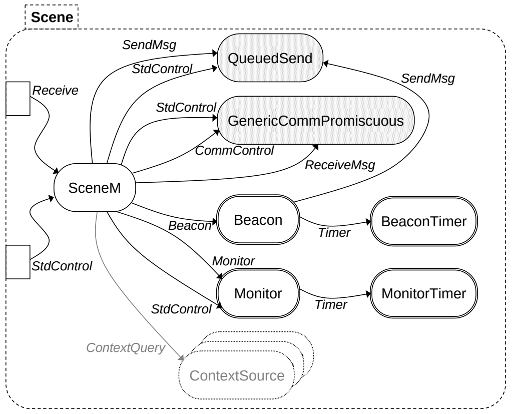


Figure 2.6: Implementation of the `scene` on sensors

The `Scene` configuration uses the ReceiveMsg interface (provided in TinyOS), which allows the component to receive incoming messages from the radio (by handling the `receive` event). Specifically, within the `Scene` configuration, the `SceneM` component handles this event. `SceneM` implements most of the logic of the scene implementation on the sensor. The structure of the messages received through this process is shown in Fig. 2.7.

While the model allows a scene to be defined by multiple constraints, a single `SceneMsg` contains only one constraint. This is a limitation of our proof-of-

```
typedef struct SceneMsg{
   uint16_t seqNo
      //message sequence number
   uint8_t metric
      //constant selector of metric
   uint8_t costFunction
      //constant selector of cost function
   uint16_t metricValue
      //current calculated value of metric
   uint16_t threshold
      //cutoff for metric calculation
   uint16_t previousHop
      //the parent of this node
   uint8_t maintain
      //whether the query is persistent
   uint8_t data [(TOSH_DATA_LENGTH-11)]
      //the query
}
```

Figure 2.7: `SceneMsg` definition

concept implementation that will be removed in a more mature implementation. The `SceneMsg` contains a sequence number that uniquely identifies the message. The sequence number is a combination of the unique client device id and the device's sequence number. This allows a receiving node to differentiate between `scenes` for different client applications. A message contains two constants that instruct the `SceneM` component in processing the message: the metric (e.g., `SCENE_DISTANCE` or `SCENE_LATENCY`) and the path cost function (e.g., `SCENE_DFORMULA` or `SCENE_MAX`). The use of constants to specify the metric and cost function makes the implementation a little inflexible because the set of metrics must be known *a priori*, but the approach prevents messages from having to carry code. Future work will enable this automatic code deployment. The `metricValue` in the `SceneMsg` carries the previous node's calculated value for the specified metric and is updated at the receiving node. In the case of a `scene` based on location, the `metricValue` may be the location of the source node, while in the case of a metric based on end-to-end latency, the `metricValue` may be the aggregate total latency on the path the message has traveled. The `previousHop` in the `SceneMsg` allows this node to know its parent in the routing tree and enables scene maintenance. The `maintain` flag indicates if the query is long-lived (and therefore whether or not the scene should be maintained). Finally, `data` carries the application message (i.e., the `Query`).

Fig. 2.8 shows how a scene message is processed at a receiving node. When `SceneM` receives a message it has not received before (based on the message's unique sequence number), it determines whether the node should be a member of the sceneby calculating the node's metric value based on the metric and path cost function. Because these fields are constants, `SceneM` can lookup their meanings in a table and determine how to calculate the new metric value. Depending on the metric, this may require *ContextSources*, which provide relevant data for calculating a node's value. For example, a hop-count based scene requires no context source; `SCENE_HOP_COUNT` indicates that the local metric value is "1" and the path cost function `SCENE_SUM` indicates that this value should be added to the `metricValue` carried in the message. On the other hand, `SCENE_DISTANCE` indicates the local metric value is the node's

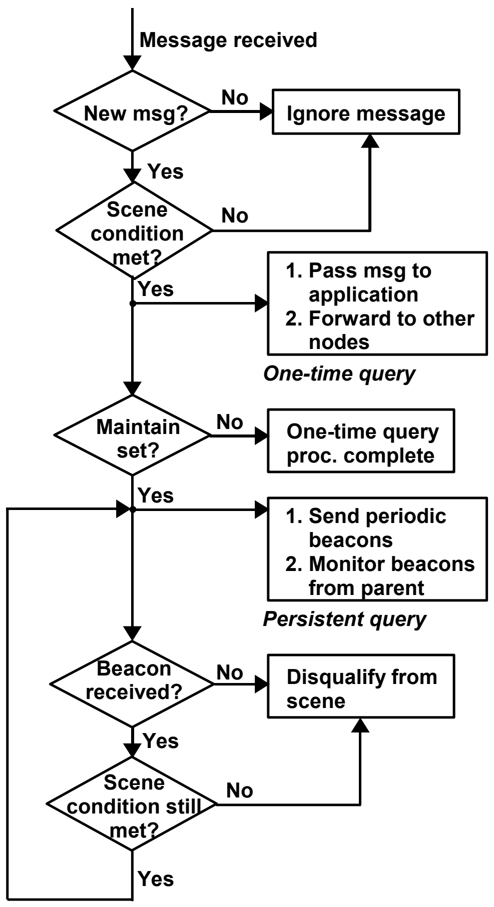


Figure 2.8: Scene construction flowchart

location, which is implemented as a *ContextSource* that stores the node's location. If necessary, context values are retrieved from the designated ContextSource through the query command in the ContextQuery interface. If the metric demands a context source that the node does not provide (e.g., the local device has no location sensor), the device is not considered a part of the scene. When the necessary context values have been retrieved, the costFunction from the message is invoked. For example SCENE_DFORMULA calculates the distance between this node and the originating node (whose location is carried in the metricValue). The newly calculated value for the metric is compared against the value of the threshold in the SceneMsg. If the new value does not satisfy the threshold, then this node is not within the scene and the message is ignored.

If this node is within the scene, the message is forwarded to allow inclusion of additional nodes. The node replaces the previousHop field with its node id. The metricValue field is populated according to the type of the metric; in the case of SCENE_HOP_COUNT, the metricValue is the total number of hops traversed so far (as calculated by adding one to the previous metricValue), while in the case of SCENE_DISTANCE, the metricValue is always the location of the originating node. This new message is broadcast to all neighbors (using TOS_BROADCAST_ADDR as the destination). The node also passes data to the application (through the Receive interface shown in Fig. 2.6).

The scene also needs to be maintained in the case of a persistent query. If the maintain flag is set, then SceneM must monitor changes that may impact the node's membership. For example, if the scene is defined by relative location and the user is walking through the network, as he moves away from a sensor, the sensor will need to be removed from the scene. The scene implementation on the sensors uses a Beacon module to transmit periodically to other nodes. As the Monitor component (described next) detects changes in the metric value, the value is updated (through

SceneM) and reflected in the beacons sent to neighbors. In addition, SceneM must monitor incoming beacon messages from the parent. Such messages are received in SceneM and passed to the Monitor. The Monitor uses beacons from the parent, information about the scene (from the initial message), and information from the context sources to monitor whether the node remains in the scene. In addition, the MonitorTimer requires that the node has heard a beacon from the parent at least once in the last three beacon intervals. If either the parent has not been heard from or the received beacon pushes the node out of the scene, the Monitor generates an event for SceneM that ultimately ceases the node's participation in the scene, including signaling the application to cancel its interactions with the client device.

2.4 An Example Scene

In this section, we tie the code that the application developer writes through the scene communication protocol to what happens on the sensors. We follow a query from the application developer's hands into the network and back. Within this section, we use an example application drawn from the first responder domain that assumes personnel deployed on a dangerous site that may contain smoke clouds. Specifically, we assume a first responder would like to periodically receive any reading within 5m that can be delivered in less than 15ms in which the level of combustion products in the air exceeds 3% obscuration per meter.

Step 1: Declare a Scene. This first step uses the interface described in Section 2 to declare a scene. For example, in a first responder deployment, the code in Fig. 2.9 defines a scene that includes every sensor (not just those measuring smoke conditions) within 5m of the declaring device and with response latency less than 15ms.

Fig. 2.10 shows the nodes that will fall in the scene, if a message is distributed to them.

```
Scene s = new Scene({new Constraint(Scene.SCENE_DISTANCE,
                                           Scene.SCENE_DFORMULA,
                                           new IntegerThreshold(5)},
                        {new Constraint(Scene.SCENE_LATENCY,
                                           Scene.SCENE_MAX,
                                           new IntegerThreshold(15)} ) ;
```

Figure 2.9: First responder scene construction

Step 2: Create a query. The next step is performed by the application developer using the `Query` data type in conjunction with the `Scene` instance just created. In our example, the developer creates a `Query` with two `Constraints`. For simplicity, we assume the application-level processing uses constraints similar to those used in scene definitions. In actuality, the scene protocol can deliver application messages of any form to all scene members, including, for example, middleware messages in a sensor network middleware [32]. In our example `Query`, the first of the constraints requires the sensor used to support a smoke detector. The second constraint limits the sensors that respond to the query to only those that measure a smoke condition of more than 3% obscuration per meter. The code used to construct this `Query` is shown in Fig. 2.11.

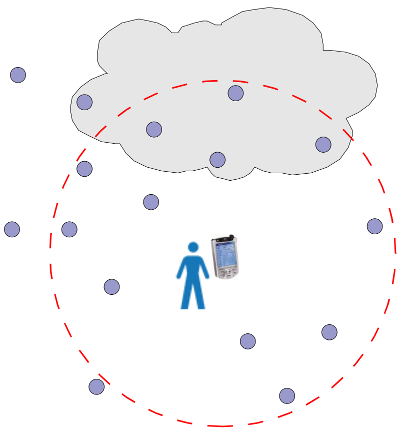


Figure 2.10: The scene

```
Query q = new Query(new Constraint(‘‘Sensor’’, Query.EQUALS_OPERATOR,
                                     ‘‘Smoke’’),
                     new Constraint(‘‘Measurement’’, Query.GT_OPERATOR,
                                     ‘‘3’’)} );
```

Figure 2.11: Example first responder query construction

Every sensor in the scene that has a smoke sensor periodically evaluates the query, but a sensor will only send a response to the client if and when the smoke condition sensed exceeds 3% obscuration per meter. After creating this `Query`, the application developer dispatches it using the previously created scene.

Step 3: Construct and Distribute Protocol Query. The scene implementation transforms the application's request into a protocol data unit for the scene. The resulting message carries the information about scene membership constraints *and* the data query. By its definition, the communication protocol ensures that the data query is delivered to only those sensor nodes that satisfy the scene's constraints.

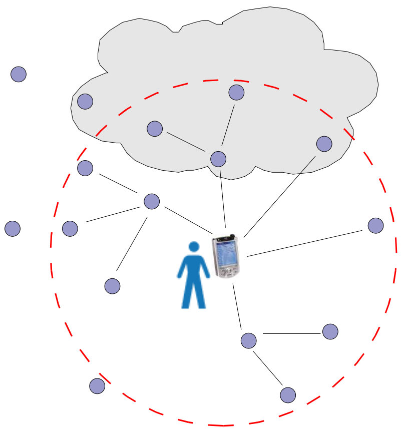


Figure 2.12: The query dissemination tree

Thus, exactly the sensors within 5m and with a latency less than 15ms will receive the query. The query propagation stops once a node is reached whose distance from the user exceeds 5m or whose latency exceeds 15ms. Fig. 2.12 shows the dissemination tree; nodes within the dashed circle now know they are scene members.

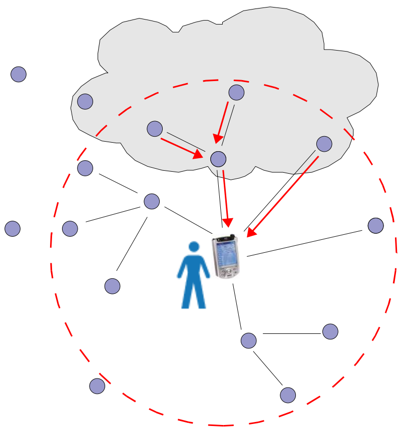


Figure 2.13: The responses from scene members

Step 4: Scene Query Processed by Remote Sensor. When the communication protocol running on a remote sensor receives and processes a scene message, if it determines that the node lies within the scene, it passes the received message to the application. In our example first responder scenario, our simple application layer sends periodic responses

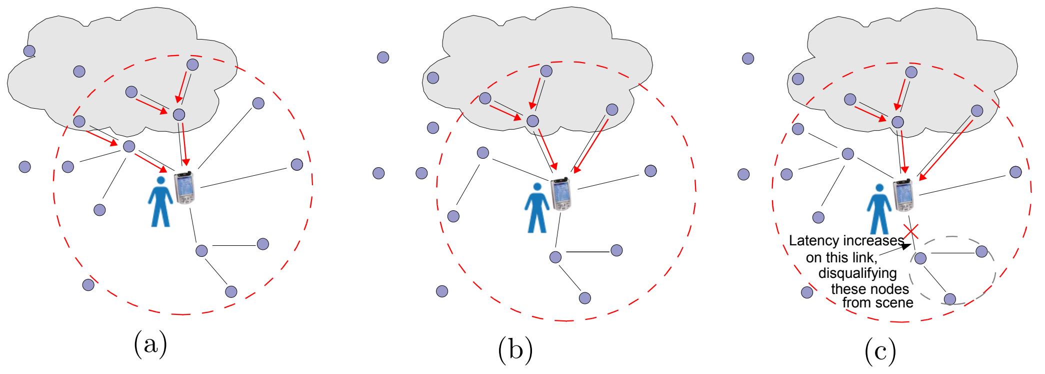

(a)
(b)
(c)
Latency increases on this link, disqualifying these nodes from scene

Figure 2.14: Dynamics within a scene. (a) The smoke cloud moves, changing responses; (b) The client moves, changing scene membership; (c) The latency increases on one link, changing scene membership.

to the client if the value exceeds 3% obscuration per meter. These responses propagate using basic multihop routing.

In Fig. 2.13, the red arrows indicate the return paths these sensors use to return query responses to the client device. Since the first responder demands periodic results so he can monitor changes in smoke density on a site, the scene must be maintained in the face of changes. If the smoke condition is not originally greater than the threshold, the node only starts responding if the 3% obscuration per meter level is reached. Fig. 2.14(a) shows that this set of responding nodes may change when the smoke cloud moves. When a node is no longer in a scene, the scene communication implementation on that node creates a null message that it sends to the application layer to ensure that it ceases communication with the client device. Other changes in the network topology or physical environment can also cause scene changes. In our example, if the node's distance from the user exceeds 5m due to client mobility (Fig. 2.14(b)), or the latency to a node on the path exceeds 15ms (Fig. 2.14(c)), the scene membership may have to be recalculated. As demonstrated in the figures, this may cause nodes to be removed from the scene or new nodes to be added to the scene. We note that these increases in distance and latency could be due to many factors, including the movement of the sensor nodes themselves. We do not necessarily assume a static sensor network; we assume that sensor nodes are usually static, but they may move occasionally.

Step 5: Result Received by Client Device. After propagating through the underlying communication substrate, query replies will arrive at the client device's sensor network interface. At the client device, the result is handled by the `scene` implementation on the sensor and passed into the Java implementation. This implementation demultiplexes the request and hands it back to the appropriate application through the `ResultListener` that was provided as part of dispatching the query to the `scene`. At this point, control for this query reply transfers back to the client's application and its `ResultListener`, which handles the query's result (or queries' results if multiple matches existed). For persistent queries, as more results arrive, the same process occurs for each received result.

As this example has demonstrated, the `scene` abstraction seamlessly supports client mobility within an immersive sensor network. The abstraction automatically adjusts the application's view of data in response to changes in the network or the physical environment. This context-awareness is essential to pervasive computing applications that rely on localized interactions in large-scale networks. In the next section, we provide some performance characterizations of the protocol implementing the `scene` abstraction to show that it provides good scalability and overhead in such resource-constrained networks.

2.5 Evaluation

Our implementation is written for TinyOS [22] in the nesC language [16]. We have chosen TinyOS since it is a widely used, open source operating system for sensor networks. TinyOS and programs for TinyOS are written in nesC, which is optimized for the resource constraints of sensor nodes. In nesC, an application consists of modules wired together via shared interfaces to form configurations.

We have created an implementation of a basic scene protocol that we have evaluated using TOSSIM [39], a simulator that allows direct simulation of code written for TinyOS. We note that this implementation of scene construction is purely reactive; that is, scenes are created on demand, in response to an application's request for a new scene.

2.5.1 Simulation Settings

In generating the following results, we used networks of 100 nodes, distributed in a 200 x 200 foot area, with a single client device moving among them. We used two types of topologies: 1) a regular grid pattern with 20 foot internode spacing and 2) a uniform random placement. While the sensor nodes remained stationary, the client moved among them according to the random waypoint mobility model [26] with a fixed pause time of 0. To model radio connectivity of the nodes, we used TOSSIM's empirical radio model [58], a probabilistic model based on measurements taken from real Mica motes [12]. In all cases, as the client moves, the scene it defines updates accordingly. In the different simulations, the client either remains stationary or moves at 2mph, 4mph, or 8mph (e.g., 4mph is a brisk walk). In these examples, scenes are defined based on the number of hops relative to the client device, ranging from one to three hops. Other metrics need to be easily exchanged for hop count; we selected it as an initial test due to its simplicity. A final important parameter in these measurements is the beacon interval. Recall that the beacon interval is the specified amount of time over which each node monitors beacons from its parent in the routing tree to maintain its own membership in the scene for continuous queries. If the node does not hear from its parent during that beacon interval, it disqualifies itself from the scene. The length of the beacon interval needs to be such that the scene protocol can keep up with client mobility. We have currently set the beacon interval to be inversely proportional to client speed. Since this approach requires shared global knowledge, it is not reasonable, and this is not how beacon intervals will actually be assigned in the future. Future work will investigate how to dynamically determine the optimal beacon interval, and this information can be included in the scene building packets, allowing nodes to adapt the beacon interval depending on the application's situation.

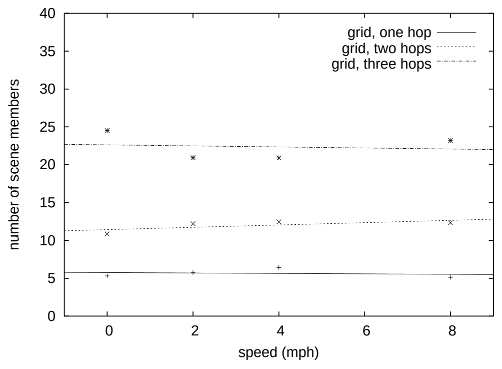
2.5.2 Performance Metrics

We have chosen three performance metrics to evaluate our implementation: (i) the average number of scene members, (ii) the number of messages sent per scene member, and (iii) the number of messages sent per unit time. We evaluate these metrics for both grid and random topologies.

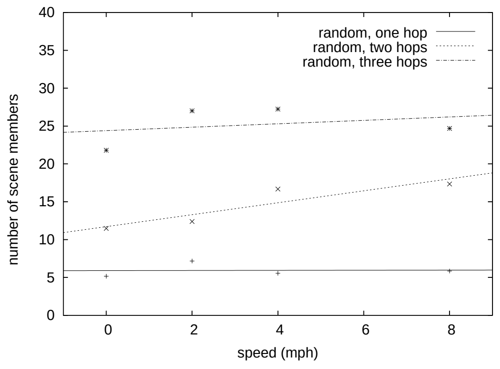
The first metric measures how well our selected beacon intervals perform. The latter two metrics measure the scalability of the scene abstraction, i.e., how the protocol will function in scenes of increasing sizes and client mobility. The number of messages sent per scene member measures a sensor node's *cost of participation*, which also estimates the potential battery dissipation for the sensors that participate in the scene (since energy expended is proportional to radio activity). The number of messages sent per unit time is a measure of the network's *average activity.* Since the scene protocol operates on-demand, activity takes place only within the scene.

2.5.3 Simulation Results

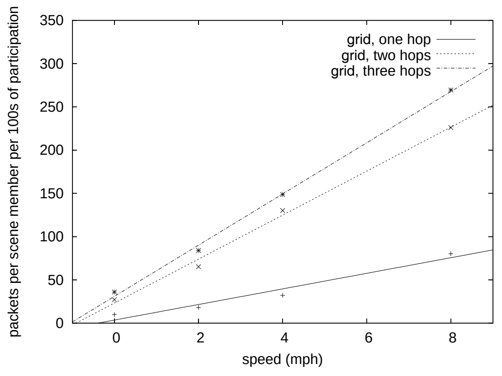
Figures 2.15(a) and (d) show the average number of scene members as a function of client device mobility and scene size for grid and random topologies, respectively. The number of scene members is almost independent of the client node's speed. This means that the device is able to accurately reach the nodes that need to be members of its scene and shows that setting the beacon frequency to be proportional



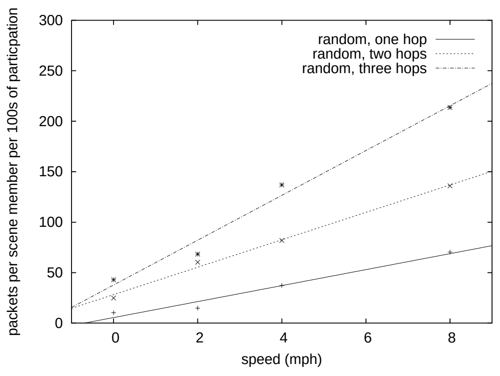
(a)



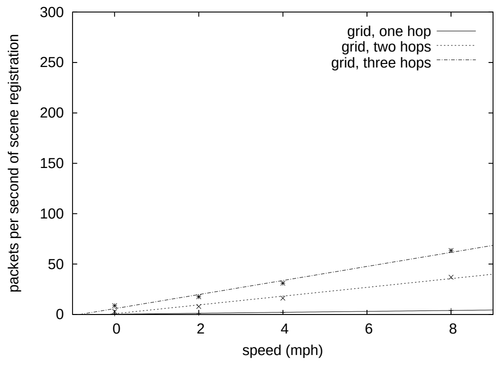
(b)



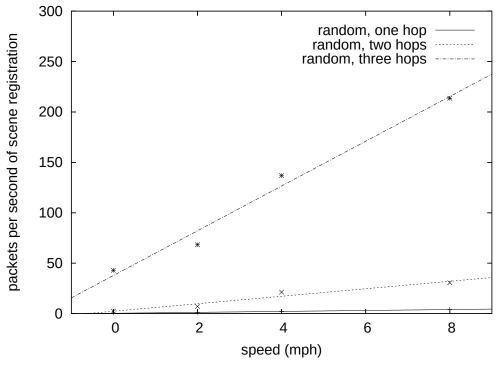
(c)



(d)



(e)



(f)

Figure 2.15: Simulation results

to the client node's speed accurately keeps track of the moving client. Setting a good beacon interval without global knowledge is an open research problem that is left for future work.

Figures 2.15(b) and (e) show the number of messages sent per scene member as a function of client mobility and scene size. Because we have set the beacon frequency to be directly proportional to the speed of the client node (e.g., if the client speed is 4 mph, beacons are sent every 0.5s, if the client speed is 8 mph, beacons are sent every 0.25s), beacons are sent more frequently as speed increases, yielding the linear relationship. This shows how the battery dissipation for each sensor that participates in the scene would scale with increasing client mobility.

Figures 2.15(c) and (f) show the number of messages sent per unit time as a function of mobility and scene size. Beacons are sent more frequently as the client node speed increases, causing more messages to be packed into a given time interval. In addition, as the scene size increases, more nodes become scene members, increasing the number of nodes that subsequently send beacon messages per unit time.

These results demonstrate that even as the scene size increases, the overhead of creating a local communication neighborhood is manageable and localized to a particular region of interest. Since the scene protocol is an on-demand communication protocol, the activity in the network takes place only within the scene. The nodes that do not satisfy the scene constraints are inactive. The average number of scene members stays constant over changing client mobility for a specified scene size.

2.6 Chapter Summary

In this chapter, we first discussed existing grouping protocols. Our goal is to support the types of pervasive computing applications that require direct, opportunistic interactions among heterogeneous devices and sensors. To this end, we created a communication paradigm consisting of an abstraction and implementing protocol. This novel protocol constructs an application's operating environment (i.e., the sensors with which it interacts) based on a specification provided by the application. The abstraction defines local, multihop neighborhoods surrounding a particular application, supports mobility by dynamically updating the scene's participants, and minimizes how much the developer must know about the underlying implementation. We defined a formal scene model, presented an implementation for the protocol, and performed simulation evaluations that demonstrate that even as the scene size increases, the overhead of creating a local communication neighborhood is manageable and localized to a particular region of interest. In the next chapter, we present a new in-network aggregation mechanism.

Chapter 3

Virtual Sensors: An Intuitive Programming Abstraction

To support ubiquitous computing environments, sensor networks will need to support localized cooperation of sensor nodes to perform complicated application-directed tasks and in-network processing to transform raw data into high-level abstract information which is not necessarily a measurement the physical sensors themselves can provide. We create an in-network aggregation model that can apply arbitrary and complex user-specified functions to different types of data available in the instrumented environment in an adaptive and decentralized manner, while minimizing the amount of data transferred over the network. In this chapter, we describe the virtual sensors model and implement some example applications. First, however, we overview some existing work.

3.1 Related Work

As the main goal of this work is to provide heterogeneous in-network aggregation in support of pervasive computing, we will overview a combination of related work that have addressed homogeneous in-network aggregation, heterogeneous aggregation, and heterogeneous in-network aggregation.

3.1.1 Homogeneous In-Network Aggregation

Several recent research efforts have focused on in-network data aggregation techniques. Projects targeted directly for sensor networks have often explored representing the sensor network as a database. Two demonstrative examples are TinyDB [44] and Cougar [61]. TinyDB includes an implementation of the Tiny Aggregation (TAG) [43] framework for data aggregation using an SQL-like language. Generally, these approaches create routing trees to funnel replies back to the root (the data requester). As data flows up the tree, it is aggregated in the network using the specified aggregation function.

The data aggregation scheme in [55] extends the types of of queries sensor networks can answer. This scheme supports queries such as approximate quantiles (median), most frequent data values, histograms, and range counting. It can also handle changing data values and continuous monitoring.

VirtuS [9] offers temporal and spatial aggregation of homogeneous data types and the transformation of a single type into another. It inserts a virtual sensor layer between the user-defined application components and the sensor driver. Thus, it can provide homogeneous readings when network nodes have different sensing boards.

3.1.2 Heterogeneous Aggregation Outside the Network

The Global Sensor Network (GSN) [2] supports an abstraction in which the user can declaratively specify XML-based deployment descriptors. SQL queries over local and remote sensor data sources can merge sensor network data. However, GSN assumes that the sensor network uses one or more dedicated computers which perform expensive data mining operations before returning results.

Semantic Streams [60] provides a service concept that allows the user to issue queries over semantic values and to extract semantic information from raw sensor data. The user does not have to specify the data or operations that should be used. However, the approach focuses on fixed sensor infrastructures such as those in homes and office buildings, where most of the sensors are powered and wired, or at most one hop away from a base station.

The major drawback of the above approaches is the fact that they do not perform the aggregation in the network.

3.1.3 Heterogeneous In-Network Aggregation

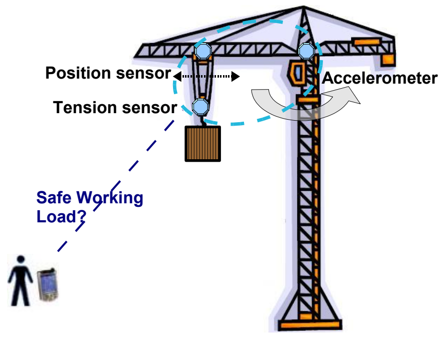
In [8], a user-specified aggregation function evaluated on a *logical neighborhood* [46] results in a *virtual node* with attributes derived from data stored on the sensor nodes in the neighborhood. This allows the abstraction of application-defined subsets of nodes into a single, logical entity. However, this approach only uses functions like average and threshold detection to trigger an event, and it is not possible to define groups to contribute to a virtual sensor based on physical properties.

The Virtual Node Layer (VNLayer) [6] provides programmable, predictable automata (or state machines), called *virtual nodes*. Several client nodes are associated with a virtual node and they reflect the behavior of the physical nodes. The client nodes are allowed to fail unpredictably, and the set of client nodes in a given region is unknown *a priori* and can change over time. However, the implementation uses Python and is intended to run on powerful mobile devices (such as PDAs or laptop computers) communicating with the 802.11 standard, and not on the resource-constrained sensor nodes. The abstraction mainly targets control, coordination, and location management applications for mobile ad hoc networks.

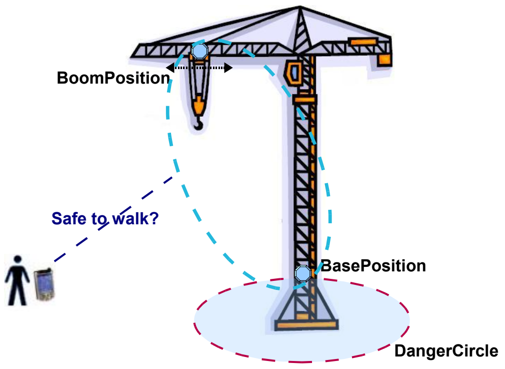
3.2 Virtual Sensors: Abstracting Data from Physical Sensors

A virtual sensor is a software sensor as opposed to a physical or hardware sensor, which builds on our initial work presented in [33, 34]. Virtual sensors provide indirect measurements of abstract conditions (that, by themselves are not physically measurable) by combining sensed data from a group of heterogeneous physical sensors. For example, on an intelligent construction site, users may desire safe load indicators on cranes that determine if a crane is exceeding its safe working capacity. Such a virtual sensor would take measurements from physical sensors that monitor boom angle, load, telescoping length, two-block conditions, wind speed, etc. [47]. Signals from these individual sensors are used in calculations within the virtual sensor to determine if the crane has exceeded its safe working load. Using fewer data types for ease of presentation, Figure 3.1(a) depicts the connection to an application-defined virtual sensor (represented by the dashed ellipse) on a tower crane. This virtual sensor uses data from three physical sensors (represented by dots). The virtual sensor aggregates the information from these sources into a higher-level reading that represents the effective load on the crane and compares this against the safe working load to warn the workers in case of danger. On the other hand, Figure 3.1(b) shows a virtual sensor that calculates a "danger circle" using readings from two different physical sensors, so that a worker walking on the site does not get hit by a moving crane boom. With the virtual sensors model, an application interacts with a combination of physical and virtual devices embedded in the environment.

The physical sensors are the components of the model that provide the physical data types required to compute the desired abstract measurement. Creating a virtual sensor requires defining a group of physical sensors that have some locality relationship (i.e., belong to a local region, where "local" is defined by some property of the network or environment). The resulting virtual sensor has an interface similar



(a) safe working load



(b) danger circle

Figure 3.1: Two different examples of virtual sensors for the construction site domain

to that of a physical sensor (from the application's perspective).

The virtual sensor hides the explicit data sources from the application, making them appear as one data source that provides the same type of interface as a physical sensor. Our approach to creating a virtual sensor's declarative specification assumes applications and sensors share knowledge of a naming scheme for the low-level data types the sensor nodes can provide (e.g., "location," "temperature," etc.). The available data types are determined by the types of sensors deployed in a network. The programmer, then, only needs to specify the following four parameters for the virtual sensor:

3.3 Virtual Sensors Model

In this section, we discuss the virtual sensors model. Different implementations of the virtual sensor model can provide in-network aggregation or out-of network aggregation, depending on the particular network's characteristics.

We will first give an overview of the model. We then detail the essentials of virtual sensor creation, which include specifying the input data types, and the resulting abstract data type. With the data types defined, the sensors that can provide these types need to be discovered; since we do not want to flood the entire network with messages for sensor discovery, we show how traditional scoping abstractions can be applied to limit the reach of discovery messages. We then use these building blocks to completely formalize the virtual sensor definition and end by describing how virtual sensors are subsequently used by applications.

3.3.1 Overview of the Model

In our model, several sensors required to supply the desired application-level data to the client application are encapsulated in a *virtual sensor*. A virtual sensor's declarative specification offers the ability to specify any desired function and leaves the discovery of physical data sources to an underlying communication layer [50]. It also provides a layer of abstraction for the application developer in comparison to directly programming in the low-level language that the sensing devices use.

The *physical* sensors are the components of the model that provide the basic data types required to compute a virtual sensor's abstract measurement. Creating a virtual sensor requires defining a group of sensors that have a defined locality relationship (e.g., they belong to a local region, where "local" is defined by some property of the network or environment). The resulting virtual sensor has an interface that, from the application or user's perspective, is similar to that of any other sensor (e.g., the physical sensors used to construct it).

3.3.2 Creating a Virtual Sensor

To create the virtual sensor, our model requires a declarative specification that describes the desired sensor's behavior without requiring the application developer to specify the underlying details of how the sensor network should support that behavior. Most importantly, the virtual sensor specification hides the explicit data sources from the application and the user, making them appear as one data source.

We assume applications and sensors share a naming scheme for data types (e.g., they share a knowledge of how to access a "location" or "temperature" data type). This implies some translation machinery deployed in the sensor network; this will be discussed in detail in Chapter 4. The available data types are determined by the types of sensors (both physical and virtual) deployed in a network. A programmer creating a new virtual sensor must then specify four components to completely specify the virtual sensor:

- *Input data types*: the types required to compute the abstract measurement. These can be provided by physical sensors or by other virtual sensors that have already been constructed and deployed. Each input type also includes the number of distinct sensors of this type that are required. This number could be *one*, *two*, *all*, etc. This allows us to differentiate between requests for "all of the concrete heat sensors" and "one concrete heat sensor."

- *Aggregator*: a generic function defined to operate over the (possibly heterogeneous) input data types to calculate the desired measurement. When the virtual sensor is deployed on a powerful client device, high-level code (e.g., written in Java) can be directly executed to evaluate the *aggregator*. In remote deployments, *aggregators* can be dispatched to the resource-constrained sensor network for evaluation.

- *Resulting data type*: the abstract type that results from evaluating the *aggregator* over the *input data types*.

- *Aggregation frequency*: the frequency with which this aggregation should be made. This determines how consistent the aggregated value is with actual conditions (i.e., more frequently updated aggregations reflect the environment more accurately but generate more communication overhead). This is similar to the sample frequency of a physical sensor; at each interval, the virtual sensor "samples" each sensor it uses and aggregates the results.

By providing these virtual sensor specifications, an application delegates physical sensor discovery to the framework that supports the virtual sensor. Therefore, if the data sources supporting the virtual sensor change over time, the virtual sensor can adapt, but the application need not notice. We next explore in more detail how data types are modeled in the virtual sensor framework, how a regional abstraction supports the acquisition of data to support the virtual sensor, and how data types are discovered in a distributed fashion.

3.3.3 Using a Virtual Sensor

Domain experts can create virtual sensors at any time. These virtual sensor definitions are then placed in the repository shown in Fig. 3.2 until an application requires the data type the virtual sensor provides. The virtual sensor's definition includes an aggregation frequency, but that does not mean that the virtual sensor sends updates according to that frequency once it has been defined. Physical sensors are capable of taking readings but usually do not do so until some application instructs them to. Similarly, virtual sensors do not send responses back until they are actually queried or used by an application.

3.3.4 Modeling Data Types Defining a Virtual Sensor

In our model, a data type may at times be provided by a physical sensor and at other times by a virtual sensor (e.g., location provided by GPS (physical sensor) or by triangulation (virtual sensor)). From the application's perspective, this is a single data type. This is mediated through a repository of data types that, for lack of a better term, we refer to as the "data type ontology." The structure and function of this repository are described in more detail later, but, at the model level, it allows

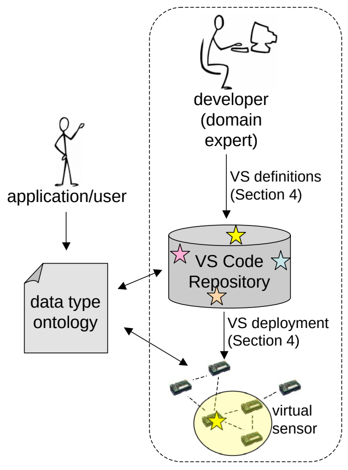


Figure 3.2: Virtual sensor architecture

the architecture to be structured as shown in Fig. 3.2.

The figure shows two points of interaction with the virtual sensors. The first is through the virtual sensor *developer*, a domain expert who can create new virtual sensors and insert them into the virtual sensor code repository. This domain expert possesses the knowledge necessary to create the domain-specific aggregator. For example, an expert in the construction domain knows the equations for computing the safe working load of the crane. Once this virtual sensor is created and made available through the virtual sensor repository, its type can be listed in the *data type ontology*. When an application or *user* wants to connect to a data type, whether it be a physical type available directly from one or more sensors in the network or a virtual sensor data type available through a piece of software defined in the virtual sensor repository, the type is listed in the data type ontology. The two double arrow lines indicate the discovery process that ensues to find the data type in the network, and, if it is not already available, to cause a virtual sensor to be deployed if possible.

This discovery process is not the focus of this dissertation; many communication mechanisms exist to support this, and the virtual sensors architecture does not commit to a particular underlying communication mechanism. We do, however, briefly discuss this issue here and from an implementation perspective in the next section.

3.3.5 Defining the Region of a Virtual Sensor

Restricting the physical devices that contribute to the resolution of the virtual sensor queries requires that a coalition of related nodes support cooperative query resolution. Protocols for establishing relative local neighborhoods have been previously explored for sensor networks [46, 50, 58, 59]. Instead of building in a dedicated neighborhood formation algorithm, the virtual sensors architecture can incorporate any one of these approaches that allows a node to form a local neighborhood around itself. For the remainder of this dissertation, we assume the particular physical devices contributing to the virtual sensor are known *a priori*; our implementation uses an approach based on on-demand distributed evaluation of proximity functions to determine the virtual sensor membership on the fly [50].

3.3.6 Formalizing Virtual Sensors

Abstractly, a virtual sensor is defined in terms of the physical devices that contribute to its construction. Assuming the physical devices available in the neighborhood described above can be represented as the set S, we can formalize the constraints on the virtual sensor definition as:

> *Given the set of hosts S in the virtual sensor's neighborhood, the required physical data types, $D_1, D_2, \ldots, D_n$, the aggregator, A, and the resulting data type, D_{res}, the virtual sensor can be formalized as:*
>
> $$D_{res} = A(D_1, D_2, \ldots, D_n), \textit{ where:}$$

$$\langle \forall D_i : 1 \leq i \leq n :: \langle \exists V : |V| = D_i.\text{count} \wedge \langle \forall s \in V : s \in S \wedge D_i.\text{type} \rhd s \rangle\rangle\rangle^{1,2}$$

In the above definition, the set V is the subset of S that defines which physical devices contribute to the virtual sensor. V may not be exactly equivalent to S in the case that extra nodes are required to provide communication connections among the nodes in V. If the construct in the last line of the definition evaluates to false, it is not possible to construct the specified virtual sensor. Recall from above that the input data types $(D_1, D_2, \ldots, D_n)$ are defined by the type of data they request and the number of independent readings of that type that are required. We assume the former is expressed as D_i.type and the latter is expressed as D_i.count. For example, the virtual sensor shown in Fig. 4.1(b) requires two data types and one sensor of each type: a single crane base position and a single crane boom position. A virtual sensor that generates the average temperature of a curing pad of concrete requires temperature values from n temperature sensors. In this case, only one D is provided, and its count value reflects the number of sensors to be polled. As we described previously, the count value included in the declaration of the virtual sensor can be a number (e.g., *one* as in the case of the crane sensors) or *all*, indicating that all matching sensors in the virtual sensor's neighborhood should be polled.

Dynamic sensor discovery, that is the discovery of the virtual sensor by the application, takes place on the basis of the virtual sensor specification. Virtual sensors provide the same interface to the user as physical sensors. The data type ontology that was mentioned above exists at this interface and defines the data types available to the application. Types of complex virtual sensors created by a domain expert can be added to this ontology. Most importantly, the application

[1] In the three part notation: $\langle$`op` *quantified_variables* : *range* :: *expression*, the variables from *quantified_variables* take on all possible values permitted by *range*. Each instantiation of the variables is substituted in *expression*, producing a multiset of values to which `op` is applied, yielding the value of the expression. If no instantiation of the variables satisfies *range*, then the value of the three-part expression is the identity element for `op`, e.g., *true* if `op` is $\forall$.

[2] The "D_i.type $\rhd$ s" construct denotes the fact that "sensor s can provide the data type specified in D_i.type."

does not have to know that it is discovering a virtual sensor instead of a physical sensor. The developer selects a data type listed in the ontology from a scene. If that data type can be provided by a physical sensor, no virtual sensor construction is necessary. Otherwise, the virtual sensor is activated and searches for supporting physical sensors in the scene.

If a virtual sensor is being used to obtain periodic responses, it needs to be dynamically refreshed. At every refresh interval specified by the virtual sensor's aggregation frequency, the virtual sensor gets new measurements from each physical sensor contributing to it and recalculates the virtual measurement. During its lifetime, some sensors contributing to a virtual sensor may deplete their battery power and become non-functional. In this case, the virtual sensor attempts to discover a new sensor that can provide the data that sensor was providing; if another such sensor does not exist in the scene, the virtual sensor fails.

Furthermore, while the sensor nodes used in ubiquitous computing environments are embedded (hence stationary), the application interacting with them runs on a device carried by a mobile user. Therefore, the dynamics associated with user movement may cause the physical sensors that comprise the virtual sensor to change (see Figure 3.3). These changes need to be seamlessly handled without revealing the underlying dynamics to the user. A dynamic maintenance is necessary that allows the user to interact directly with a changing set of local information sources and the virtual sensor needs to hide the underlying complexity from the user.

Figure 3.4 abstractly depicts n physical sensors, aggregated into a virtual sensor which can run (a) locally (on the client) or (b) remotely (in the network). The physical sensors are illustrated using circles, and the different shadings indicate their heterogeneous nature. The sensors inside the large dashed circle contribute to the virtual sensor, but only a few have been shown with arrows representing the data they send back, for ease of presentation. The virtual sensor code can either run

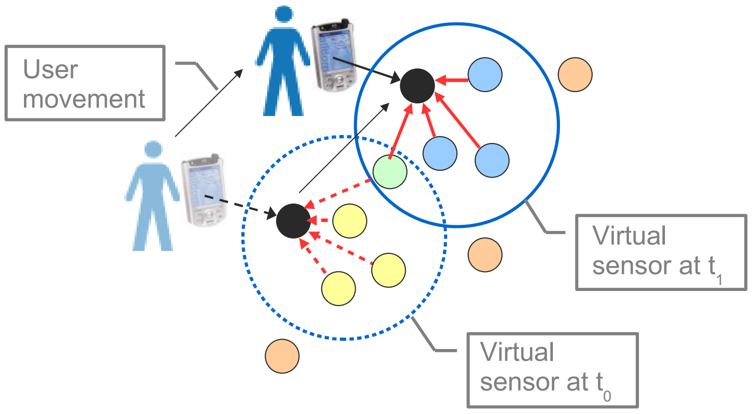
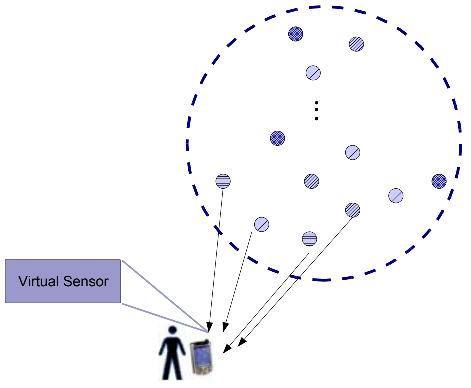


Figure 3.3: The dynamics associated with user movement

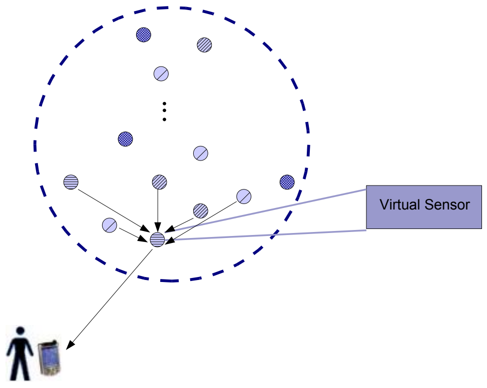
locally on the client device or be deployed to a resource-constrained sensor within the network. When deployed remotely, this code will be dynamically received by a listener on the remote sensor and executed. If all of the physical sensors are in a cluster, and that cluster is several hops away from the user's device, then it may make sense to send the virtual sensor out to the cluster. On the other hand, if each of the sensors that make up the virtual sensor is within one hop of the user, then the virtual sensor should run on the user's device. With respect to the application interface, it does not matter if the virtual sensor is deployed on the client's device or remotely in the network, but it might improve performance to use a certain option depending on the application's situation. The rest of the discussion in this chapter assumes a local deployment, while Chapter 4 assumes a remote deployment.

3.4 Example Applications

In this section, we relate two complete application examples we have fully implemented to demonstrate the use and performance of virtual sensors. To illustrate the application-independent nature and the ability to support multipurpose networks of virtual sensors, we have chosen application examples from the two different do-



(a) n physical sensors, aggregated into a virtual sensor which runs locally



(b) n physical sensors, aggregated into a virtual sensor which runs remotely

Figure 3.4: Abstract depiction of a virtual sensor that uses n physical nodes

mains, the first of which is the intelligent construction site we have introduced in Chapter 1, and the second is an aware home domain. An aware home is a residential environment that is able to obtain information about itself, as well as the locations and activities of the people situated in that environment, through embedded sensors.

Each virtual sensor specification is provided to the middleware, which translates it into two components: the virtual sensor proxy and the virtual sensor. The former runs on the user's device; the latter can be written in either Java or nesC and we will explore deploying it to a sensor in the network in Chapter 4.

3.4.1 Construction domain example

The virtual sensor we describe here allows the user to sense data of type `CraneDangerCircle` for nearby cranes. This circle represents the area near a crane where it is unsafe to walk and is centered at the base of the crane (which may move) and has a radius defined by the position of the boom (which is even more likely to move) (see Fig. 3.1(b)). As the boom moves along the crane arm, the size of the danger circle should expand and contract accordingly. An application can use this information to maintain a map of the construction site to ensure vehicles and workers are always safe and to display warnings to a worker when he enters a danger circle.

The virtual sensor programmer (a domain expert) possesses the application knowledge (more specifically, knowledge of the data types that are available and the functions that will be necessary to create the resulting data type) to create the virtual sensor. Using our middleware, this programmer can use a high-level language to create tailored sensing capabilities.

Our example virtual sensor (`CraneVS`) uses two data types available in the sensor network, selected from an ontology for the construction site: `BasePosition` and `BoomPosition`. This, too, is a simplification as these data types may themselves be the result of a virtual sensor that aggregates basic `location` data with nearby identity data (e.g., from an RFID tag) to determine that a particular location sensor is located at the base of a crane. The `CraneVS` generates abstract data of the type `CraneDangerCircle` which is delivered to the application. The code the application programmer must write to construct such a sensor looks like:

```
VirtualSensor craneVS = new VirtualSensor({new BasePosition(),
                                           new BoomPosition()},
                                           new CraneAggregator(),
                                           new CraneDangerCircle());
```

Within the application, `BasePosition`, `BoomPosition`, and `CraneDangerCircle` are data types that extend the `DataType` class. The application may have to create the `CraneDangerCircle`, but `BasePosition` and `BoomPosition` are likely to be common to the domain and therefore reusable across applications. All three data types appear in the domain's ontology. In the constructor above, new instances of the classes representing the types are constructed as placeholders.

This virtual sensor request is translated into a request for two different data types (`BasePosition`, `BoomPosition`) from the same scene. If the query initiated is persistent, these data types are encapsulated with the requested frequency and sent to the sensor(s) which send back periodic responses. The number of sensors (in this case, one of each type) that are expected to respond to the query are given in the virtual sensor specification, along with the data type.

The domain programmer must also specify the mechanics behind the aggregation within the `CraneAggregator`. This is accomplished by implementing the `Aggregator` interface and providing an implementation of the `aggregate()` method:

```
class CraneAggregator implements Aggregator {
   CraneDangerCircle aggregate(DataType[] inputs){
      int radius = Math.sqrt( (input[0].x - input[1].x) *
                              (input[0].x - input[1].x) +
                              (input[0].y - input[1].y) *
                              (input[0].y - input[1].y)   );
      return new CraneDangerCircle(input[0], radius);
   }
}
```

Our prototype implementation performs the virtual sensor aggregation in Java on the client device. The virtual sensor created to support this specification does two things. First, it calculates the radius based on the data values received from the virtual sensor's defining physical sensors. Second, it returns (in a single message) values for anything referenced in the return statement (i.e., the calculated `radius` and the (x,y) coordinates of the base of the crane, as indicated by the use of `input[0]` in the return statement). The aggregate() function encapsulates it as an object of the type the application expects (e.g., the `CraneDangerCircle` in the example). In this virtual sensor, the circle is specified by its center (the location of the crane base) and its radius.

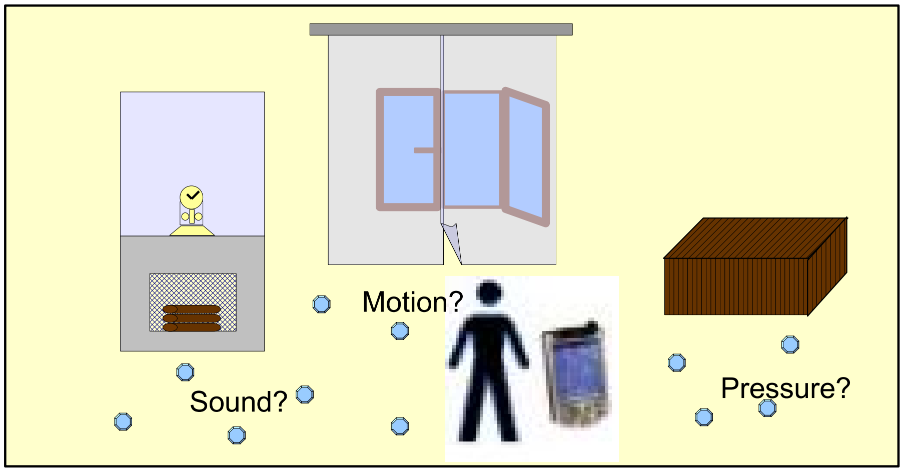


Figure 3.5: Aware home example

3.4.2 Aware home domain example

The virtual sensor we describe here allows the user to sense data of type `RoomOccupancy` for a room in an aware home. This data type indicates if there is currently a person in the room or not, based on an aggregation of sound, pressure, and motion readings obtained from the room, as shown in Figure 3.5. When a person enters or exits the room, the room's sound, pressure, and motion readings will change. An application can use this information to make decisions based on room occupancy, such as turning the lights on or off. We note that sound by itself may not be enough, since there may a clock chiming in the room, the same is true for pressure; a heavy crate may have been temporarily placed on the floor. As for motion, there may be an open window in the room which may cause a curtain to move occasionally due to wind.

Our example virtual sensor (`OccupancyVS`) uses three data types available in the sensor network in the room: `Sound`, `Pressure`, and `Motion`. The `OccupancyVS` generates abstract data of the type `OccupancyIndicator` which is delivered to the application. The code the application programmer must write to construct such a sensor looks like:

```
VirtualSensor occupancyVS = new VirtualSensor({new Sound(),
                                               new Pressure(),
                                               new Motion()},
                                               new OccupancyAggregator(),
                                               new OccupancyIndicator());
```

Within the application, `Sound`, `Pressure`, and `Motion` are data types from the aware home data ontology. The application may have to create the `OccupancyIndicator`, but `Sound`, `Pressure`, and `Motion` are likely to be reusable across applications. The domain programmer must also specify the mechanics behind the aggregation within the `OccupancyAggregator`. This is accomplished by implementing the `Aggregator` interface and providing an implementation of the `aggregate()` method:

```
class OccupancyAggregator implements Aggregator {
   OccupancyIndicator aggregate(DataType[] inputs){
      boolean occupancy = (input[0] > soundThreshold) &&
                          (input[1] > pressureThreshold) &&
                          (input[2] == motionDetected) &&
      return new OccupancyIndicator(occupancy);
   }
}
```

The above specification determines a combination of "sound," "pressure" on the floor, and "motion" to be "occupancy." We note that different houses (or different rooms in a house) could have different definitions for "occupancy."

We have implemented virtual sensors code for the two application domains discussed above (for more information on our virtual sensors code, see the Virtual Sensors Home Page [57]).

3.5 Feasibility Study

To demonstrate the feasibility of virtual sensors, we implemented and deployed a virtual sensor for an application example from the domain of the intelligent construction site [31]. In our scenario, the user requests information about the region around the base of a crane where it is unsafe to walk or drive. In this case, a virtual sensor is constructed that dynamically discovers physical sensors attached to components of a nearby tower crane (e.g., the base of the crane, the trolley along the boom, the counterweight of a crane). The virtual sensor combines the information collected from these distributed physical sensors to calculate the requested abstract data type (i.e., a danger circle calculated using location estimates from sensors attached to the crane). Once these sensors are discovered, the virtual sensor registers persistent queries on these particular sensors and remains connected to them. As the sensors generate and send updates, the virtual sensor automatically refreshes the presentation of the abstract data type and displays the changes to the application, so the worker receives a warning when entering a potentially dangerous area. This example performs on-the-fly heterogeneous data fusion from multiple sensor streams in the field.

Specifically, our application demonstration connects a set of Cricket motes [11] (a location-aware version of MICA2 Motes [45]) attached to the crane to another Cricket mote that represents a worker on the site. We simulate the situation using a Lego crane [37] (as shown in Fig. 3.6). Changes in the worker's position and crane movement occur in real time in the demonstration to show different situations.

The virtual sensor on worker's truck monitors the crane to determine if the worker is inside the danger sector by combining the crane boom position, crane counterweight position, and worker's own location. Applications can use this information to ensure workers and vehicles are always safe. A warning is displayed to the worker when he enters the danger sector (i.e., the red LED of the truck's sensor

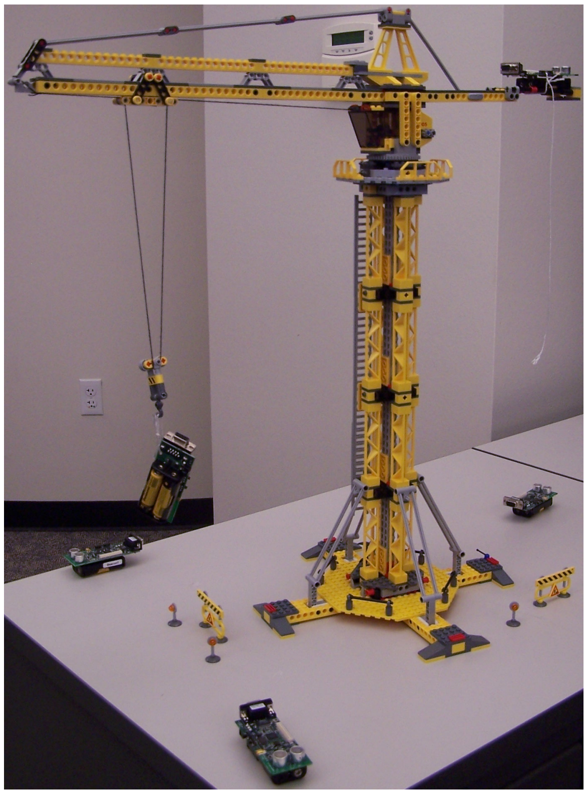
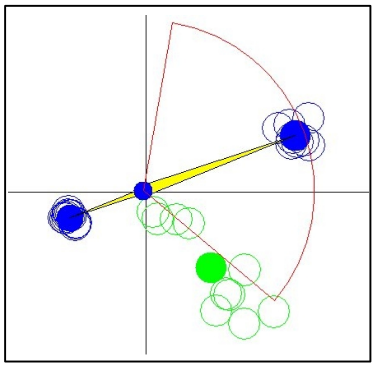


Figure 3.6: The virtual sensor on a tower crane

turns on).

We use eight Crossbow Cricket motes, four of which are used simply for obtaining the positions of the relevant sensors that contribute to the virtual sensors. These act as beacons; on a real construction site, we would use GPS satellites to obtain this information, but since GPS does not work well indoors, we had to obtain position information using Cricket motes. The Cricket mote is a location-aware version of the MICA2 mote. It uses an ultrasound transmitter/receiver for time-of-flight ranging and provides centimeter-level accuracy. One mote acts as the gateway to the laptop computer, where the data is aggregated. The remaining three measure the boom, counterweight, and worker locations.

Figs. 3.7 demonstrate safe and dangerous situations that may arise as the


(a) Worker outside danger sector (green LED on, i.e., safe)

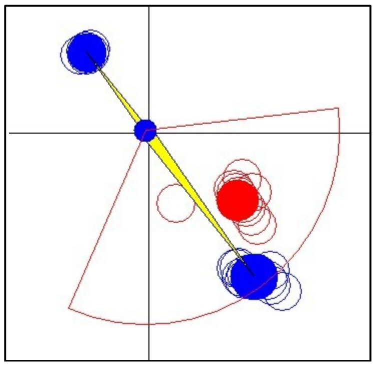

(b) Worker inside danger sector (red LED on, i.e., in danger)

Figure 3.7: The virtual sensor on a tower crane

current situation is assessed through the real-time aggregation running on the laptop computer.

3.6 Chapter Summary

In this chapter, we defined a formal virtual sensor model designed to abstract data from heterogeneous physical sensors by applying user-defined functions. We implemented two complete application examples using a local deployment to demonstrate the use and performance of virtual sensors. We demonstrated the implementation and deployment of a virtual sensor as a feasibility study. The separation of the specification of the sensing task from the sensing behavior allows a programmer to describe the behavior of a virtual sensor, without having to specify the underlying details of how it should be constructed. Virtual sensors offer a way to tailor a generic sensing environment to specific applications. This will be especially necessary as sensor networks become more widespread and general-purpose. In the next chapter, we explore two mechanisms for enabling virtual sensors to be remotely deployed on resource-constrained sensor nodes. The first is a parameterized middleware deployment that runs on the sensor nodes and accepts virtual sensor definition information within TinyOS messages. The second is to use a mobile code deployment in which the TinyOS messages directly carry code defining the virtual sensors, which can be immediately loaded into the run-time environment at the sensor node.

Chapter 4

Remote Deployment of Virtual Sensors

For virtual sensors, supporting the local deployment case shown in Fig. 3.4(a) of Chapter 3, where each of the physical data types can be independently collected from the network and then aggregated on the client device in a higher-level language (e.g., Java), was discussed in Section 3.5. In this chapter, we consider the more complicated situation depicted in Fig. 3.4(b) in which the virtual sensor is dispatched into the network to execute on resource-constrained sensing devices. In this case, the application's definition of a virtual sensor in a high-level language must be translated into a form understandable by the underlying resource-constrained sensor network. This potentially limits the expressiveness of the virtual sensors that can be specified, but it also has the potential to drastically reduce the overhead of heterogeneous aggregation in sensor networks because it reduces the number of messages that must be sent.

In this chapter, we will present two alternative styles of implementations for remote deployment of virtual sensors. We will discuss the implementation and performance characteristics for each.

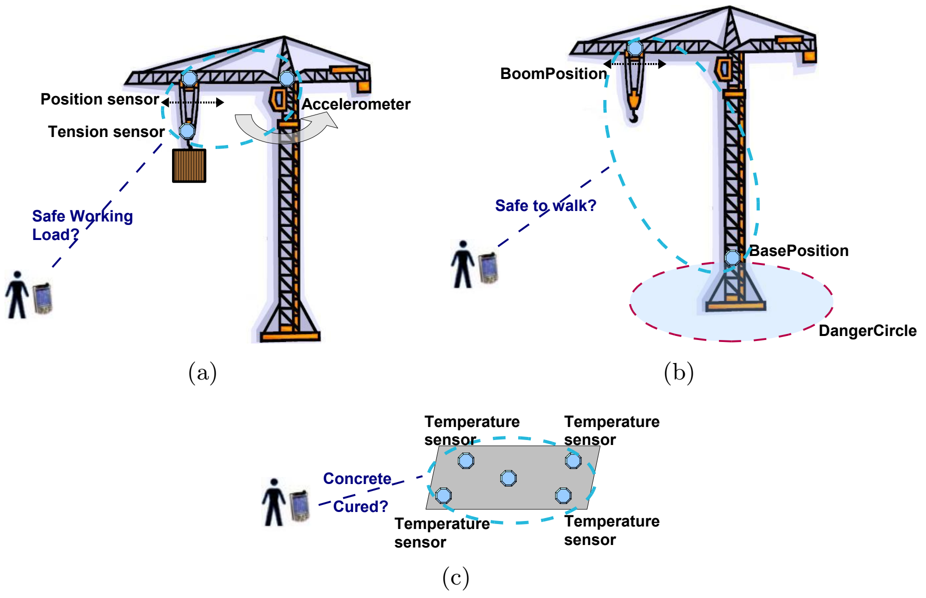


Figure 4.1: Construction site domain examples

4.1 Motivation and Problem Definition

The tiny sensing devices that support pervasive computing environments have many constraints. Limited energy is a major concern, and it is therefore essential to reduce applications' communication overhead. The amount of communication each sensor must perform can be reduced through the use of in-network aggregation, so that not all data has to be relayed back to a requesting node; instead, the necessary result can be calculated en route through node cooperation. Existing solutions generally provide only homogeneous aggregation. However, emerging user-level applications will leverage capabilities from RFIDs, smart cell phones, sensors from different vendors, etc. that provide vastly varying data types. Heterogeneous data aggregation is important in combining this data into a single abstract measurement.

It will often be most efficient in terms of communication to perform such aggregation remotely, in the network, as opposed to collecting all of the available data and processing it offline. This task is significantly more complicated than this simple statement implies; remotely deploying complex aggregation requires being able to quickly load new functionality on remote, resource-constrained sensors. In addition, sensor networks in support of pervasive computing must be reusable. Current efforts create application-specific solutions, but the future will see multipurpose networks deployed to support numerous and changing applications. The cost of physically visiting each sensor to reprogram it is prohibitive, and therefore the ability to remotely reprogram sensors to tailor them to particular applications will be essential. Some existing approaches attempt to perform the remote code deployment [5, 14, 38], but they have limitations with respect to how applications can tune the distribution of the new code, what the code can do, and in which situations mobile code can be used, as discussed in Chapter 3.

As sensor networks become widespread in their support of a variety of applications, the utility of an abstraction that addresses these challenges can be demonstrated through many examples. We draw examples from the *intelligent construction site*, where potentially mobile sensors embedded in equipment and structures can support safety and management applications [20]. We introduce three examples and revisit them throughout the remainder of the chapter.

Crane safe load sensor. On an intelligent construction site, users may desire the cranes to have safe load indicators that determine if a crane is exceeding its capacity. A safe load indicator would rely on measurements from physical sensors that monitor boom angle, load, telescoping length, two-block conditions, wind speed, etc. [47]. Using fewer data types for ease of presentation, Fig. 4.1(a) depicts the connection to an application-defined safe-load sensor (represented by the dashed ellipse) on a tower crane that uses data from three physical sensors (represented by dots). The safe-load sensor aggregates the information from these sources into a higher-level reading that represents the effective load on the crane and compares this against the crane's known safe working load.

Crane danger circle sensor. A danger circle represents the area near a crane where it is unsafe to walk. It is centered at the base of the crane (which may move) and has a radius defined by the position of the boom (which is even more likely to move). Fig. 4.1(b) shows a software sensor that calculates such a "danger circle" using two different physical sensors. As the boom moves along the crane arm, the size of the danger circle should expand and contract accordingly.

Aggregate concrete cure sensor. A construction site supervisor coordinating a team of workers spread throughout the site may want to have an aggregate view of the concrete cure conditions over a concrete slab in the site. Fig. 4.1(c) shows an aggregate concrete cure sensor that would allow the supervisor to be alerted when the concrete is ready for use. While the first two examples provide heterogeneous aggregation, this third application demonstrates homogeneous aggregation.

4.2 Parameterized Middleware Approach

In this section, we present the parameterized middleware implementation of the virtual sensors abstraction and evaluate it.

4.2.1 Implementation

In this section, we describe the parameterized middleware implementation of the virtual sensors abstraction [29]. We provide the domain expert with a Java application programming interface and a set of operations that can be combined to specify complicated virtual sensor behavior. The middleware that runs on the sensors is written for TinyOS [22] in nesC [16] and programmed onto Crossbow MICA2 motes [12]. Our implementation code is available at [51]. Virtual sensor definitions are encapsulated in TinyOS messages and distributed to sensor nodes. Instead of sending new code when the developer wants to inject a new virtual sensor, a message

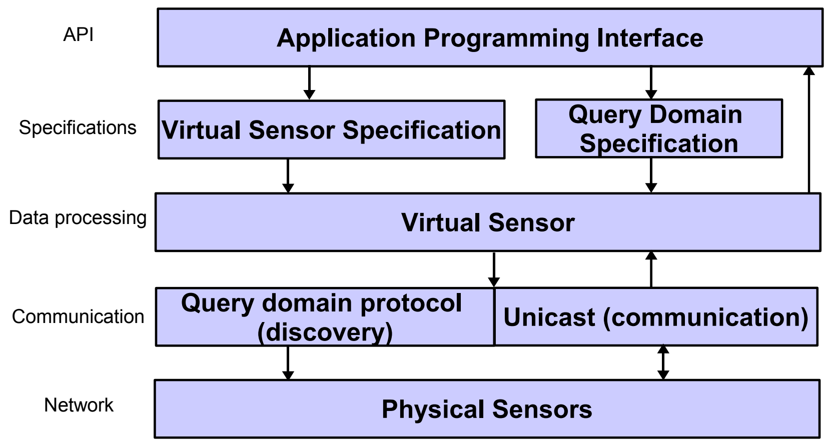

API
Application Programming Interface
Specifications
Virtual Sensor Specification
Query Domain Specification
Data processing
Virtual Sensor
Communication
Query domain protocol (discovery)
Unicast (communication)
Network
Physical Sensors

Figure 4.2: Simplified object diagram for the virtual sensor middleware

is sent that contains the parameters for specializing a generic piece of code already present within the middleware. This significantly reduces the communication and computational costs associated with loading new functionality but also decreases the flexibility of virtual sensor definitions. We evaluate these tradeoffs in later sections. The messages are described in more detail later in this section; we first describe how the developer perceives and interacts with the virtual sensors abstraction.

Application Programming Interface

Fig. 4.2 depicts the object diagram of the middleware from the developer's perspective. The application developer uses this API to specify virtual sensors, and the middleware then handles the necessary sensor network communication on behalf of the application. The virtual sensor constructed as a result of these specifications uses proximity functions [50] for sensor discovery. These query domains allow the application developer to specify required relationships among all participants in the virtual sensor (e.g., all physical sensors contributing to a virtual sensor must be within a given distance of each other). Once these groups are created, the infrastructure uses a combination of groupcast and unicast communication to maintain

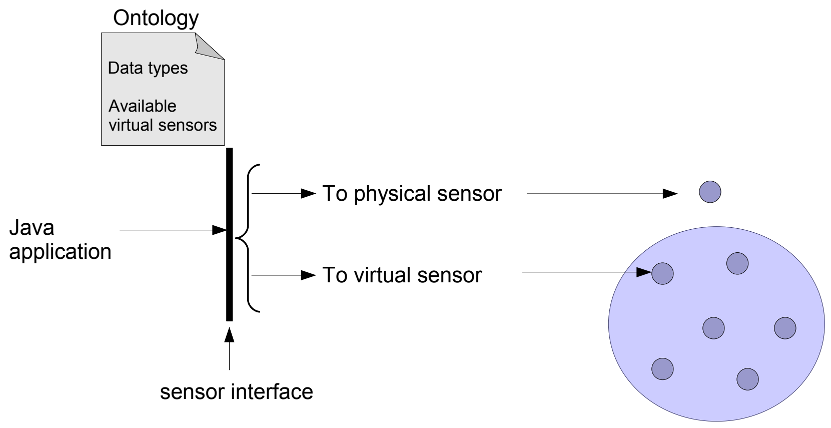


Figure 4.3: Virtual sensor architecture

the virtual sensors and to aggregate and retrieve data.

The `VirtualSensor` object shown in Fig. 4.2 is maintained at the client device and keeps a list of live queries. This allows a single virtual sensor to support queries from multiple applications in the same way that a single physical sensor can provide data for multiple applications. A virtual sensor is deployed only when there are active queries, and the information from the virtual sensor is accessed on-demand.

Fig. 4.3 shows the distributed virtual sensor architecture. The application perceives a single interface to access to both physical and virtual sensors. The developer's high-level code (written in Java) interfaces with the sensors using a data type ontology that includes built-in general-purpose data types (e.g., temperature, location, angle, etc.) and provides a straightforward mechanism for inserting additional types. When the application needs to query the network for a data type that is not directly provided by the physical sensors, the developer constructs and deploys a virtual sensor using his knowledge of the available data types (as expressed in the ontology). The application subsequently queries this virtual sensor directly, in the same manner that it queries other physical sensors.

When defining new virtual sensors, the developer needs to specify the low-

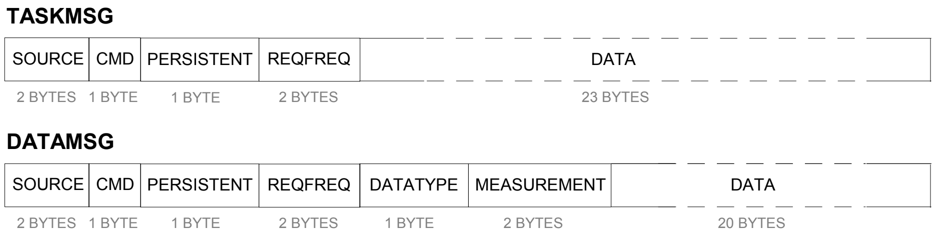


Figure 4.4: Packet formats for the TaskMsg and DataMsg messages

level data types and the relationships required to compute the desired abstract measurement. This gives the developer the maximum amount of flexibility in specifying new data types and how they should be formed while hiding the underlying implementation details from the end user, lending a powerful virtual sensor implementation.

Virtual Sensor Tasking

Domain experts define virtual sensors using a high-level programming language. Our middleware translates these definitions into a specific format that is placed in TinyOS messages and distributed within the sensor network. Specifically, to create a virtual sensor and to obtain data from it, we use two different packet formats, as shown in Fig. 4.4. The first of these, the task message, contains a field that describes the operations the virtual sensor should perform. Since RPN (Reverse Polish notation, also known as postfix notation) [54] provides a compact representation that is machine-readable, we define the following grammar based on RPN for the data field of the virtual sensor task messages:

$$E \rightarrow E\ E\ \text{bin\_op} \mid \text{datatype\_all set\_op} \mid \text{datatype \# set\_op} \mid$$

$$\text{datatype datatype bin\_op} \mid \text{datatype \# bin\_op} \mid \text{datatype un\_op}$$

where:

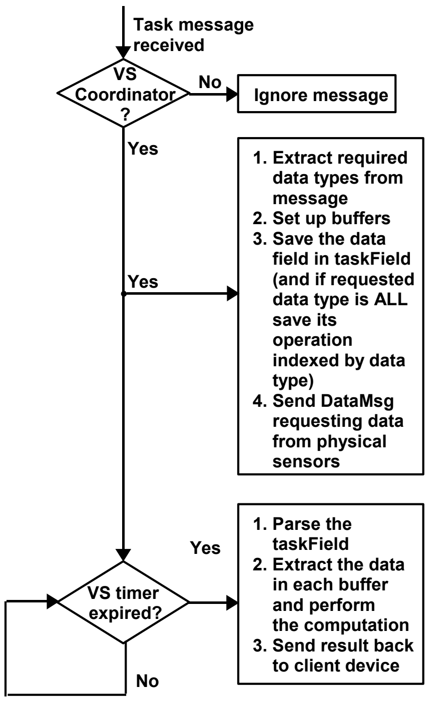
$$\text{bin\_op} = \{+, -, *, \div, \text{POW}, <, \ldots\}$$
$$\text{set\_op} = \{\text{MIN}, \text{MAX}, \text{SUM}, \text{AVG}, \ldots\}$$
$$\text{un\_op} = \{\text{SQRT}, \text{COS}, \text{SIN}, \ldots\}$$

and datatype refers to single instances of sensor types available in the network, datatype_all indicates a desire to apply a set operation to *all* available readings of the indicated type, and # refers to an integer number (between 0 and 10).

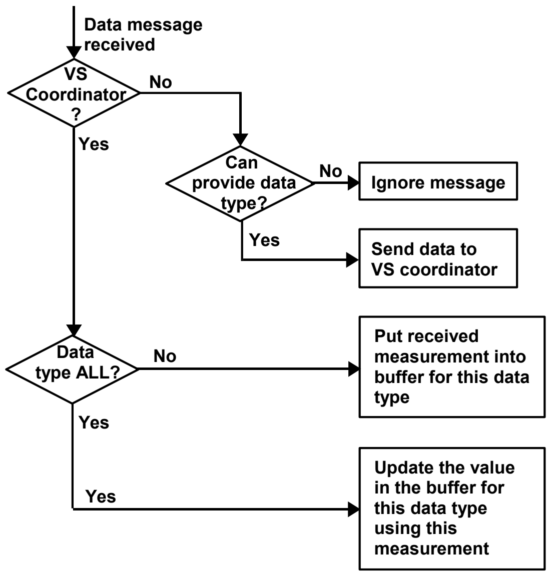
This grammar defines unary operators (e.g., the square root), binary operators (e.g., addition), and set operators (e.g., minimum). As defined in the grammar above, data types associated with an expression are interpreted differently depending on the type of operator. For example, a task message containing the data [Temperature_all AVG] instructs the created virtual sensor to collect all of the available temperature readings and average them, while [Temperature 5 AVG] instructs the created virtual sensor to collect five temperature readings and average them.

Numbers, data types, and operators make up the alphabet of this grammar, and we map each one to a reserved number to enable its transmission in a data message:

- *0-10*: Numbers
- *11*: "All" (reserved to support additional functionality)
- *12-31*: Operators (set operators and binary operators)
- *32-143*: Data types (with the ability to specify the number of distinct sensors that should provide a value)
- *144-255*: Data types "all" (this requires all sensors within the proximity that


Task message received
VS Coordinator ?
No
Ignore message
Yes
1. Extract required data types from message
2. Set up buffers
3. Save the data field in taskField (and if requested data type is ALL save its operation indexed by data type)
4. Send DataMsg requesting data from physical sensors
Yes
VS timer expired?
Yes
1. Parse the taskField
2. Extract the data in each buffer and perform the computation
3. Send result back to client device
No

(a) TaskMsg received


Data message received
VS Coordinator ?
No
Can provide data type?
No
Ignore message
Yes
Send data to VS coordinator
Yes
Data type ALL?
No
Put received measurement into buffer for this data type
Yes
Yes
Update the value in the buffer for this data type using this measurement

(b) DataMsg received

Figure 4.5: Flowcharts for message reception

can provide this data type to respond)

Virtual Sensor Operation

The middleware running on the virtual sensor coordinator is tasked to provide an abstract measurement through the necessary calculations that are stated in a task message. To support the task, the virtual sensor coordinator sets up buffers for each data type required for the computation and sends data messages to the contributing physical sensors, asking for their data. When the data comes back from all the physical sensors, the virtual sensor coordinator performs the operations in the order stated in the task message (and if the query is continuous, this happens at every periodic virtual sensor query interval).

Figs. 4.5(a) and (b) show the message processing flowcharts for the task messages and the data messages, respectively.

Since the goal is to collect data and process it in the network as it arrives, and since we use an RPN-based grammar for the task field, we use a stack to perform the calculations while parsing the task message (shown Fig. 4.5(a)). Until the end of the task field is not reached, the task field is parsed and all the operations are performed in order:

- If the next symbol is an operand (a number or a data type), we place it (the number itself or the measurement for the data type pulled from the buffer for that data type) on top of the stack.
- If the next symbol is an operator, we remove the top elements from the stack, perform the operation, and place the result on top of the stack.

The stack reduces the amount of data the virtual sensor coordinator must store. It also allows a "running" operation that is computed on the fly and does not need to store historical data. To summarize, the task message and the way it is parsed allows any combination of binary and unary operations as well as comparison operations on any type of data to transform it to an abstract data type, giving the application developer flexibility and the ability to retask sensor nodes.

4.2.2 Evaluation

We evaluate four characteristics of the remote virtual sensor: the abstraction's expressive power, the complexity of applications created for the middleware, the complexity of the middleware itself, and the communication overhead required to support the abstraction.

Table 4.1: Example virtual sensor definition messages. For each example, we give the abstract virtual sensor definition first written in standard form, then in RPN. Finally, we give the definition written in the hexadecimal form. in which it is actually transmitted in our prototype. The latter depends on the (static) list of types available for our application domain and on the static mapping of operators to values as shown in Section 4.2.1.

Application	**Virtual Sensor Definition**
Safe Working Load	$(\texttt{ActualLoad} * \texttt{BoomLength}) < \texttt{MaximumLoadMoment}$ $\texttt{ActualLoad BoomLength} * \texttt{MaximumLoadMoment} <$ $\boxed{\texttt{24}}\boxed{\texttt{25}}\boxed{\texttt{0E}}\boxed{\texttt{26}}\boxed{\texttt{16}}\boxed{\texttt{0 ...0}}$
Danger Circle	$\sqrt{(\texttt{BasePositionX} - \texttt{BasePositionY})^2 + (\texttt{BoomPositionX} - \texttt{BoomPositionY})^2}$ $\texttt{BasePositionX BasePositionY} - 2\ \texttt{POW BoomPositionX BoomPositionY} - 2\ \texttt{POW} + \texttt{SQRT}$ $\boxed{\texttt{20}}\boxed{\texttt{22}}\boxed{\texttt{0D}}\boxed{\texttt{02}}\boxed{\texttt{10}}\boxed{\texttt{21}}\boxed{\texttt{23}}\boxed{\texttt{0D}}\boxed{\texttt{02}}\boxed{\texttt{10}}\boxed{\texttt{0C}}\boxed{\texttt{11}}\boxed{\texttt{0 ...0}}$
Curing Concrete Pad	$\texttt{AVG}(\texttt{CureRate}, \texttt{ALL})$ $\texttt{CureRateAll AVG}$ $\boxed{\texttt{90}}\boxed{\texttt{15}}\boxed{\texttt{0 ...0}}$

Expressive Power

The expressive nature of the virtual sensor allows applications to request arbitrary functionality, making it flexible and useful in interacting with intermittently connected sensor nodes. An application can create a virtual sensor directly tailored to its needs. The application can also dynamically change the virtual sensor by sending a new virtual sensor task message.

We demonstrate the virtual sensor abstraction's expressive power using the three application examples introduced in Section 4.1: a) a crane's safe working load; b) a crane's danger circle; and c) a curing concrete pad. We have deployed each application on four MICA2 motes running the virtual sensor middleware. Upon receiving a TaskMsg from outside the network, one of the nodes becomes the virtual sensor coordinator, gathers the required data from the other nodes, adds its own data (if applicable), performs the calculation, and sends the aggregate result back to the client device. Table 4.1 shows the data field of the TinyOS message for each virtual sensor. For each virtual sensor, it takes an average of 1.031 seconds to perform the calculation on the MICA2 mote, but this includes the one second it takes for the virtual sensor to acquire all the data from the other motes contributing to the virtual sensor.

Reduction of Application Complexity

Without the virtual sensors middleware, an application developer must construct the entire behavior by hand. This includes querying each physical node and locally aggregating the returned data values.

The more straightforward way of coding afforded by the virtual sensor simplifies the programmer's task. Instead of being concerned with low-level communication details, the programmer delegates complex coding to the middleware. The developer relies on the programming interface described in the previous section to provide the aggregation behavior which can be optimally placed in the sensor network. In addition, the potential ability of the middleware to dynamically determine the best location in the network for the remotely deployed virtual sensor can significantly reduce the communication overhead. For example, instead of retrieving the sensor measurements from each node, then applying a threshold to each retrieved value locally, filtering can be performed at the remote virtual sensor, limiting the data that must be sent back. In a local virtual sensor deployment with four motes, each mote would have to send a message back to the client device, resulting in a total of 4 messages, but in a remote deployment, only one message has to be sent all the way back to the client device (i.e., the result from the virtual sensor coordinator).

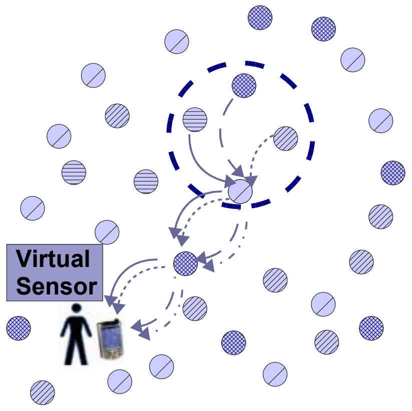
Middleware Complexity

We have quantified the overhead introduced by our middleware in terms of the memory footprint. The MICA2 platform has 128KB of program memory (ROM) and 4KB of primary memory (RAM). The virtual sensor code (written in nesC) for the crane danger circle application yields the following memory footprint for TinyOS and MICA2 motes:

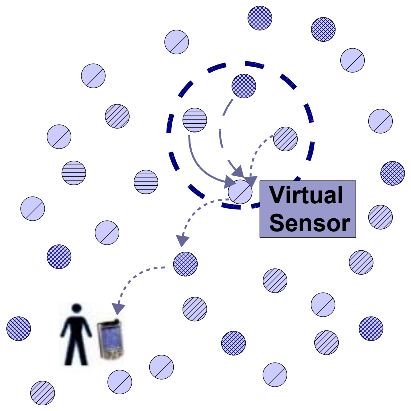
- local deployment: occupies 15,872 bytes in ROM and 544 bytes in RAM.
- remote deployment: occupies 17,234 bytes in ROM and 1,303 bytes in RAM.

In comparison to computing virtual sensor measurements locally, remotely deploying the virtual sensor does increase the memory footprint due to the need to have a generic middleware deployed that can accept tailored virtual sensor definitions. The middleware empowers remote nodes in the network to perform complicated operations without having to send as much data back to a client node.

In terms of message size, the virtual sensors middleware fits all virtual sensor definitions into 29 byte TinyOS messages. More complex virtual sensor definitions



(a) Physical sensors aggregated into a virtual sensor which runs locally



(b) Physical sensors aggregated into a virtual sensor which runs remotely

Figure 4.6: Two different types of virtual sensor deployments

than those given as examples here may become lengthy, requiring more than one 29 byte message to distribute the virtual sensor tasking information; support for this multiple message format is saved for future work.

Reduction in Communication Overhead

The amount of communication in a sensor network directly relates directly to the nodes' energy expenditure and consequently to network lifetime. The key aspect of the communication overhead is the amount of data (in bytes) that must be sent. Without the virtual sensor, each data value from the remote physical sensors must be sent back to the client device for processing.

Let a virtual sensor, V, require data from n distinct physical sensors, P_1, P_2, ..., P_n. Let us assume that P_n is the physical sensor that is closest to the client device from the set $\{P_1, P_2, ..., P_n\}$ and choose that as the virtual sensor coordinator. Assuming that P_n is h_n hops away from the client device, and that $\{P_1, P_2, ..., P_{n-1}\}$ are h_1, h_2, ..., h_{n-1} hops away, respectively, from the virtual sensor coordinator (see Fig. 4.6 for a simple example, where $n = 4$, $h_4 = 2$, and

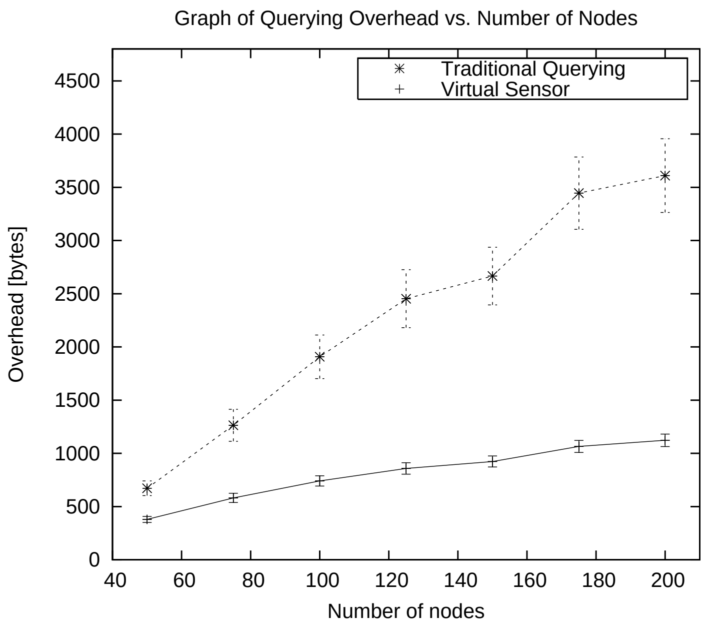
$h_1 = h_2 = h_3 = 1$):

- A local deployment of such a virtual sensor would take $h_1 + h_2 + ... + h_{n-1}$ transmissions for all the other physical sensors to send their data to node n. Then, for this node to forward all of these data values and its own value would require an additional $n \times h_n$ transmissions, resulting in a total of $nh_n + \sum_{i=1}^{n-1} h_i$ transmissions.

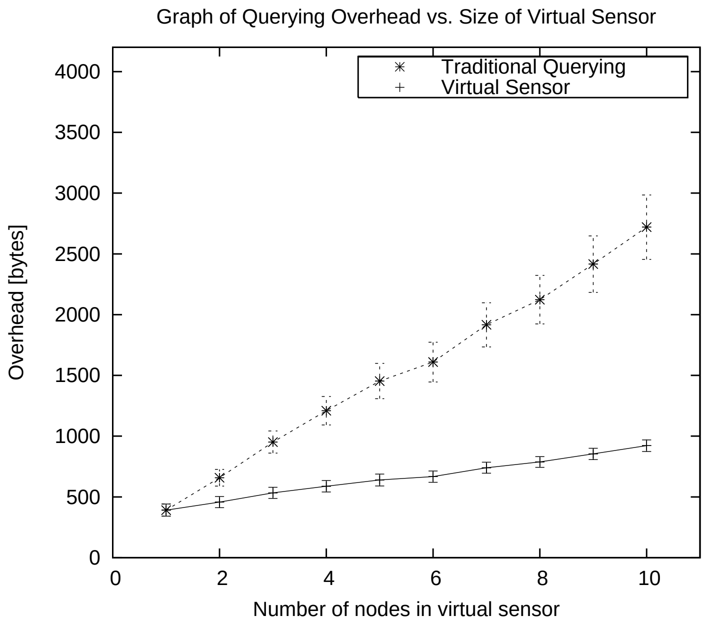
- In the case of a remote deployment, it would take $h_1 + h_2 + ... + h_{n-1}$ transmissions for all the other physical sensors to send their data to virtual sensor coordinator. Since the virtual sensor coordinator aggregates all this data before sending it to the client device, it would take only h_n transmissions thereafter to relay this data to the client node, resulting in a total of $h_n + \sum_{i=1}^{n-1} h_i$ transmissions.

Given the above, remotely deployed virtual sensors can reduce communication overhead significantly, especially as the number of physical sensors contributing to a virtual sensor increases or as the number of hops between a client device and the virtual sensor coordinator increases.

Fig. 4.7 show the measured communication overhead as a function of the number of nodes in the network and of the number of nodes participating in the virtual sensor (i.e., the size of virtual sensor), respectively. Using a virtual sensor in a network of increasing size reduces the overhead compared to a traditional querying approach. As the virtual sensor size increases, the difference in the amount of overhead between the approach using a virtual sensor and the traditional querying approach increases significantly.



(a)



(b)

Figure 4.7: Reduction in communication overhead when using a virtual sensor

4.3 Mobile Code Approach

In this section, we describe an implementation of the virtual sensors abstraction that uses a mobile code deployment in which the network messages directly carry code defining the virtual sensors, which can be immediately loaded into the run-time environment at the sensor node. The user interface that runs on the client device is written in Java. The virtual machine that runs on the sensors is written for TinyOS [16] in nesC [16].

We will describe this dissemination process in more detail later in this section; we first discuss some mobile code mechanisms that are available and give a justification for our choice.

4.3.1 Mobile Code in Sensor Networks

Maté [38] is a virtual machine designed for sensor networks that divides event-based applications into capsules that are flooded across the network. Maté assumes that only one application runs in the sensor network, so all the sensor nodes execute the same code. Deluge [24] reprograms the whole sensor network by flashing the instruction memory of each mote and supports multihop reprogramming. However, it takes a long time to transfer the new image (which is the same for all sensors). Also, sending a large image to every mote in a wireless sensor network is limited by the sensor hardware and the multihop nature of connections. Like Maté, Deluge does not allow multiple applications to be installed on nodes. Agilla [14] is an agent-based sensor network middleware that allows applications to inject agents that migrate intelligently to carry out the applications' tasks. SensorWare [5] injects mobile executable scripts. Unlike Maté and Deluge, it allows multiple applications to be running on the same node. However, SensorWare requires platforms with 1 MB of program memory and 128 KB of RAM, and the scripts support only weak mobility in the network.

We have chosen Melete [63] to distribute the code that the virtual sensor nodes need to run. Melete is based on the Maté virtual machine. However, unlike Maté, Melete can run concurrent applications on one sensor node and allows code injection to a subset of the network (not necessarily the whole network). Melete uses the TinyScript language it inherits from Maté to form the grouping of this subset.

4.3.2 Implementation

In this section, we describe the mobile code implementation of the virtual sensors abstraction. The user interface that runs on the client device is written in Java. The virtual machine that runs on the sensors, Melete, is written for TinyOS in nesC and simulated in TOSSIM. Virtual sensor tasks are written in TinyScript, encapsulated in Melete code capsules, and distributed to sensor nodes. These code capsules are described in more detail later in this section; we first describe how the developer perceives and interacts with the virtual sensors abstraction. We note that the application programming interface is the same as the one discussed in Section 4.2.1.

Virtual Sensor Architecture

In this particular implementation, the virtual sensor consists of a coordinator and subordinate nodes (the data providers).

1. **Virtual Sensor Coordinator:** One sensor node forms a local neighborhood around itself. It is tasked with finding the necessary data sources and performing operations on the data that comes back from all the physical sensors that are involved in this computation. In this sense, the virtual sensor coordinator actually acts as a centralized in-network processor when it is deployed remotely.

2. **Virtual Sensor Data Providers:** The virtual sensor data providers are physical sensors that have been discovered by the virtual sensor coordinator and are able to provide the various data types that the virtual sensor coordinator needs to be able to compute the final data type.

Virtual Sensor Tasking

Domain experts define virtual sensors using a high-level programming language. Our middleware translates these definitions into a specific format that is placed in TinyOS messages and distributed within the sensor network.

Tasking of the virtual sensor coordinator and the sensor nodes that provide data to it can be implemented in two ways:

1. The user device sends all the code to the virtual sensor coordinator, and the virtual sensor coordinator sends the code to the virtual sensor data providers, *OR*

2. The user device sends the code to the virtual sensor coordinator, the virtual sensor coordinator discovers the virtual sensor data providers, and then the user device sends the code to the virtual sensor participants.

Due to the constraints of Melete, we use the first approach in this implementation. To support virtual sensor tasking, we use the group formation functionality of Melete to group the data providers into one group. This way, the code they need to run can be sent directly to them. The code that defines the tasks that the nodes perform is encoded in a TinyScript script.

Virtual Sensor Operation

The virtual sensor coordinator is tasked to provide an abstract measurement through the necessary calculations that are stated in the mobile code. To support the task, the virtual sensor coordinator retrieves data from the contributing physical sensors. When the data comes back from all the physical sensors, the virtual sensor coordinator performs the operations in the order stated in the mobile code (and if the query is continuous, this happens at every periodic virtual sensor query interval).

Table 4.2: Melete group bitmasks

Group 4	Group 3	Group 2	Group 1	Group 0
---	---	---	---	---
16	8	4	2	1

Table 4.2 shows the binary bitmasks for the Melete group specifications. As examples, a bitmask of 4 corresponds to Group 2, a bitmask 5 corresponds to Group 0 and Group 2 (both), and a bitmask of 16 corresponds to Group 4. Melete uses groups to support concurrent reuse of sensor nodes by different applications. As a subset of sensor nodes that host an application, the group is a logical concept. Groups may overlap, their members may be connected through multihop connections, and they are dynamically adjustable.

In Melete, Node 0 is a member of every group and Group 0 is the group that includes all the nodes in the network. By convention, new groups are formed in the "once" context of Group 0. A maximum of 16 groups is possible. For a node, the number of simultaneously associated groups is constrained by its RAM capacity (e.g., up to 5 applications on TelosB node). The node checks a constraint predicate and the answer determines if should then join or leave a group in question. The constraint check and action can be moved into a timer context. A sensor node can specify a set of target groups as the receiver of a broadcast. This group specification is encoded in the broadcasted message.

To support our virtual sensor abstraction, we build the Melete VM with the following five contexts:

- Reboot: Runs the code after the VM reboots
- Timer0: Runs the code when timer0 fires (need to set timer with "reboot reboot.txt" (e.g., `settimer0(10)`)
- Timer1: Runs the code when timer1 fires (need to set timer with "reboot reboot.txt" (e.g., `settimer1(15)`)
- Once: Runs the code once (when it installs) (e.g., `start timer0`)
- Broadcast: Runs the code when the mote receives a packet broadcast by another mote with the bcast() function

The function, bcast(), broadcasts to the one-hop neighbors of a node. As an example, bcast(4, data), where 4 is the bit mask of Group 2. Also, this function triggers the broadcast context of the target group to run. Melete also has a send() function that routes to Node 0, but this is done using the routing layer below the VM and the VM has no control over it. So, we use the bcast() function to send data between the VMs running on the different nodes. Since send() sends only to mote 0 (i.e., it sends a buffer to the base station), we use this function in the very end when the aggregated data is ready to be sent to the client device.

4.3.3 Evaluation

With respect to the mobile code implementation of the virtual sensors abstraction, we evaluate two characteristics: expressive power, complexity of the virtual machine running on the nodes. As for the complexity of applications created for the middleware and the communication overhead required to support the abstraction, arguments made in Section 4.2.2 apply here as well.

Expressive power

The expressive nature of the virtual sensor allows applications to request arbitrary functionality, making it flexible and useful in interacting with intermittently connected sensor nodes. An application can create a virtual sensor directly tailored to its needs. The application can also dynamically change the virtual sensor by sending new virtual sensor code to the virtual sensor coordinator.

We demonstrate the virtual sensor abstraction's expressive power using one of the application examples introduced in Section 4.1, a crane's safe working load. In TOSSIM, code capsules are deployed onto motes running Melete. A part of the Python script that runs the application is shown in Fig. 4.8. We note that `cl` is an instantiation of the `ScripterCommandLine` that is used for injecting scripts. The TinyScript code that is injected to form the group of virtual sensor data providers is shown in Fig. 4.9. Then, the timer is set using the code shown in Fig. 4.10. The virtual sensor data providers broadcast their readings when the timer expires (the code is given in Fig. 4.11). When the virtual sensor coordinator receives these data broadcasts, it aggregates them and sends the result back to the client device (the code is given in Fig. 4.12).

In this example, we sample the temperature sensor; this can be easily extrapolated to a crane safe load application, where Node 0 is the user device, Node 1 is the virtual sensor coordinator, Node 2 sends the actual load, Node 3 sends the boom length, and Node 4 sends the maximum load moment. Or, for a homogeneous aggregation application, Nodes 2-4 could all send the cure rate of a concrete slab.

Virtual Machine Complexity

We have quantified the overhead introduced by the virtual machine in terms of the memory footprint. The MICA2 platform has 128KB of program memory (ROM) and 4KB of primary memory (RAM). The Melete virtual machine built to support virtual

```
# File name:  VSscript.py
# Step 1:  Create group 2 (which includes VS data sources)
cl.inject(‘‘group_datasource.tsc’’, ‘‘once’’, 0)
# Step 2:  Make group 2 (VS data providers) sense and broadcast readings
# bcast(integer, buffer):  bitmap for target groups and a data buffer.
# Broadcasts data buffer for target group to local radio neighbors.
# Receiving broadcast triggers broadcast context of target group to run.
cl.inject(‘‘set_timer.tsc’’, ‘‘once’’, 2)
cl.inject(‘‘broadcast_reading.tsc’’, ‘‘once’’, 2)
# Step 3:  Make VS coordinator (Node 1) aggregate data heard from broadcast
of data providers
# A program injected in the broadcast context runs when a node receives a
# broadcast (sent by another node with the bcast() function).
cl.inject(‘‘broadcast_heard.tsc’’, ‘‘broadcast’’, 0)
```

Figure 4.8: VSscript.py Python script

sensor functionality occupies 42,798 bytes in ROM and 7,417 bytes in RAM (for MICA2 with 5 contexts: Reboot, Once, Broadcast, and two Timer contexts), so we had to restrict our evaluations to simulation in TOSSIM. However, when the Melete virtual machine is built to run on TelosB motes, the first application with these 5 contexts occupies approximately 3.1 KB memory, and each additional application occupies approximately 1.46 KB, making 5 concurrent applications possible [63].

Since, a MICA2 only has 4KB of RAM, if an application exceeds approximately 3,500 bytes of RAM (stack space will be used as well and stack space varies), it will run out of memory and the mote will simply not run. If the application uses a lot of stack space, the program may not be able to run with anything over 3,000 bytes of RAM.

In comparison to computing virtual sensor measurements locally, remotely deploying the virtual sensor does increase the memory footprint due to the need to have a virtual machine running on each node that can accept virtual sensor tasks in code capsules. The virtual machine allows remote nodes in the network to perform complicated operations without having to send as much data back to a client node.

```
!!  File name:  group_datasource.tsc
!!  If data source (i.e., Node 2, 3, or 4), join group 2.
buffer data;
private nodeid;

data[] = temp(); !!  set data equal to temperature reading
nodeid = id();
if (nodeid > 1 and nodeid < 5) then
   joingrp(2);
   led(2); !!  turn on green LED
else
   leavegrp(2);
end if

bclear(data);
```

Figure 4.9: group_datasource.tsc script

```
!!  File name:  set_timer.tsc
!!  Set timer
buffer data;
private nodeid;
settimer0(5);
```

Figure 4.10: set_timer.tsc script

The Melete virtual machine deploys code using 128-byte Maté code capsules. Complex scripts defining virtual sensor tasks require more than one 128-byte code capsule to distribute the virtual sensor tasking information; they are sent as sequential chunks of code capsules.

4.4 Qualitative Comparison of the Two Approaches

In the previous two sections, we explored two different mechanisms for enabling virtual sensors to be remotely deployed on resource-constrained sensor nodes. The first was a parameterized middleware deployment that ran on the sensor nodes and

```
!!  File name:  broadcast_reading.tsc

!!  Can broadcast a buffer over the radio with the bcast function.
!!  If the VM hears a broadcast packet, it triggers the Broadcast handler,
!!  which can retrieve the buffer with the bcastbuf function.
!!  Virtual sensor data providers (Nodes 2, 3, and 4) broadcast their readings

private reading;
private nodeid;
buffer buf;

nodeid = id();
if (nodeid > 1 and nodeid < 5) then
   reading = int(temp()); !  set reading equal to int version of temperature
   bclear(buf);
   buf[0] = id();
   buf[1] = reading;
   !  Triggers the broadcast context of the target group to run
   !  bitmask 1 = Group 0; bitmask 4 = Group 2
   bcast(1, buf); !  1 means we broadcast to Group 0
end if
```

Figure 4.11: broadcast_reading.tsc script

accepted virtual sensor definition information within TinyOS messages. The second approach used a mobile code deployment in which code capsules defining the virtual sensor tasks were sent to sensors and these capsules could be immediately loaded into the run-time environment at the sensor node.

If we compare the two approaches in terms of the message size, the parameterized middleware approach requires 29-byte TinyOS messages, while the mobile code approach requires 128-byte code capsules (Maté's default code capsule size, CAPSULE_SIZE, is 128).

In terms of simplifying programming for the application developer, the parameterized middleware approach has all the necessary code written in nesC; this code is optimized for the specific application by extracting parameters from the

```
!!  File name:  broadcast_heard.tsc
!!  VM can broadcast a buffer over the radio with the bcast function.
!!  If VM hears a broadcast packet, it triggers the Broadcast handler,
!!  which can retrieve the buffer with the bcastbuf function.
!!  Node 1 received broadcast from virtual sensor data providers
!!  Single-hop communication primitive used to locally aggregate
!!  sensor readings
private val;
buffer received;
private nodeid;
buffer buf;
nodeid = id();
if (nodeid = 1) then
   received = bcastbuf();
   val = int(received[1]);
   led(4); !  turn on the yellow LED
   bclear(buf);
   buf[0] = 5;
   send(buf);
end if
```

Figure 4.12: broadcast_heard.tsc script

TinyOS message. In the mobile code approach, the virtual machine runs injected TinyScript scripts.

As for memory requirements, the parameterized middleware occupies 17,234 bytes in ROM and 1,303 bytes in RAM (for the crane danger circle application) on a MICA2 mote. The virtual machine built for a MICA2 in the mobile code approach occupies 42,798 bytes in ROM and 7,417 bytes in RAM (built with 5 contexts to support the crane safe load application).

In terms of code storage, the parameterized middleware approach requires all the code to be stored on each node; this code is optimized and used as required by the application. In the mobile code approach, a basic virtual machine is stored on all nodes, and new code is uploaded as needed. The first approach stores some unnecessary application code that may not be required for a specific application, but has to reside there to support multiple applications.

In the mobile code approach, since nodes that run the Melete VM can only broadcast to their immediate neighbors, the virtual sensor providers have to be within one hop of the VS coordinator. The parameterized middleware approach does not have such a requirement.

4.5 Chapter Summary

In this chapter, we investigated the remote deployment of virtual sensors through two different mechanisms: middleware parameterization and mobile code. We have shown through our evaluation that it is feasible to support remote deployment through either one of these two approaches that accept tailored virtual sensor definitions to create new aggregation functionality in a sensor network on-demand. In combination with a neighborhood definition scheme such as [50], the virtual sensor provides a powerful abstraction that not only has the potential to reduce the complexity of programming pervasive computing networks but can also increase the efficiency with which such networks can respond to applications' demands. Furthermore, the virtual sensor abstraction is not application-specific. In the next chapter, we combine the scenes model from Chapter 2 and virtual sensors model from Chapter 3 in a middleware implementation.

Chapter 5

The DAIS Middleware

As the ultimate goal of this dissertation, we create a middleware, which integrates abstractions such as the scene and the virtual sensor to provide a complete system that simplifies the development of ubiquitous computing applications.

Systems designed to address the specific challenges posed by sensor networks and/or ubiquitous computing have recently been a topic of research discussions. Existing work has highlighted several design tenets that a middleware for wireless sensor networks must adhere to [62], and our middleware platform attempts to follow these guidelines. Other projects have also undertaken similar efforts, and we highlight a few of these systems.

5.1 Middleware Approaches

It is largely recognized that constructing applications for ubiquitous computing environments is a significant undertaking. Several approaches have made strides in simplifying the kind of programming necessary for immersive networks. This section provides a thorough comparative investigation into these existing techniques, from middleware solutions to toolkits and programming languages.

One strong example of a middleware for ubiquitous computing, Gaia [53], introduces *active spaces* as a programmable environment by encapsulating the heterogeneity of devices that are located in them. It abstracts user data and applications into a *user virtual space* that has a dynamic mapping to the resources in the current environment. Users always have their virtual space available, even as they move across different active spaces. Furthermore, they can simultaneously interact with multiple devices, dynamically reconfigure applications, pause and resume applications, and use context attributes to program application behaviors [53]. However, this model assumes a centralized system structure which is in direct opposition to the goal of deploying large numbers of applications over a widely dispersed immersive sensor as described in the previous section.

In contrast, projects targeted directly for sensor networks more directly address the desire to reduce computational and power requirements and to operate in a more distributed fashion. Two demonstrative examples that have explored representing the sensor network as a database are TinyDB [44] and Cougar [61]. Ubiquitous computing applications often require the use of many nearby sensors, ultimately aggregating these disparate pieces of data into a cohesive piece of information for the application or user. Therefore, despite the fact that moving data across the network in approaches such as TinyDB and Cougar still requires centralized algorithms, they have much to offer in support of ubiquitous computing applications in immersive sensor networks.

Other approaches have focused more specifically on the programmability of ubiquitous computing environments. VM* [36] is a virtual machine approach that can scale software components depending on the constraints of each device. This allows application developers to better manipulate unpredictable environments with a wide variety of devices but has limitations in that the virtual machine must know about the applications in advance to be able to optimize resource usage. Tiny-

GALS [7] allows programmers to represent applications in terms of relatively high-level components which are subsequently synthesized into the low-level, lightweight, efficient programs that are deployed on the nodes. This eases the programming task but does not allow arbitrary applications to access the immersive sensor network and immediately start to use it. MiLAN [21] aims to enable applications to control the network's resource usage and allocation optimally to tune the performance of an entire sensor network through the definition of application policies that are enacted on the network. MiLAN tries to maximize network lifetime as well as meet the application's quality-of-service requirements. While such approaches are highly beneficial when the application is known and the networks are relatively application-specific, they do not map well to immersive sensor networks where the nodes must be able to service a variety of unpredictable applications.

More generalized approaches attempt to provide integrated suites of tools that enable simplified programming of sensor networks. For example, EmStar [17] provides a suite of libraries, development tools, and application services that focus on coordinating microservers (e.g., sensing devices with computational power equivalent to a PDA). However, EmStar functions only on Linux-based platforms such as the Stargate. The Sensor Network Application Construction Kit (SNACK) [18] consists of a set of libraries and a compiler that makes it possible to write very simple application descriptions that specify sophisticated behavior using components written in nesC (the TinyOS programming language that runs on sensor motes). While EmStar and SNACK are programming environments for individual nodes, Agilla [14] is an agent-based middleware that allows applications to inject agents into the sensor network that coordinate through local tuple spaces and migrate intelligently to carry out the applications' tasks. Multiple autonomous applications can run simultaneously over the sensor network. However, Agilla shares tuple spaces transiently only on the same node and different nodes have different tuple spaces.

One approach that does map well to the operational picture shown in Figure 1.1(b) is TinyLime [13], a tuple space based middleware that enables mobile computing devices to interact with sensor data in a manner decoupled in both space and time. Applications create tuple templates to subscribe for data that is of interest to them. The tuple spaces of a pair of devices are temporarily federated whenever the devices are within a single hop of one another. TinyLime allows client devices to connect to sensors available in the immediate environment but does not enable multihop communication or aggregation. Another adaptation of the Lime model, TeenyLIME [10], uses the abstraction of a shared tuple space that contains the data of the local device and its one-hop neighbors. Since TinyLIME targets sensor networks where users with mobile devices request data from sensors immediately around them, the applications are deployed on the client devices and the sensing devices are only data producers without any tuple spaces. On the other hand, TeenyLIME applications are deployed directly on the sensing devices which have their own tuple spaces and play an active role in distributed coordination.

State-centric programming [40] mediates between an application developer's model of physical phenomena and the distributed execution of sensor network applications. It uses the notion of collaboration of groups to abstract common patterns in application-specific communication and resource allocation. Furthermore, it takes a signal processing and control theory approach, where application developers write applications as algorithms for state update and retrieval, with input supplied by dynamically created collaboration groups. Consequently, these programs are more invariant to system configuration changes and the resulting software is more modular and portable across multiple platforms. However, state-centric programming has been implemented and evaluated only on the Pieces simulator (built in Java and Matlab), which can not simulate certain important features of wireless communication such as message collision.

The Abstract Task Graph (ATaG) [3] methodology provides system-level support for architecture-independent sensing application development. ATaG uses a combination of imperative and declarative programming styles and has a data-driven program flow. For example, in an environment monitoring application, it allows the periodic computation and logging of the maximum pressure in the system, and the periodic monitoring of temperature. However, it cannot combine these two different types of data to arrive at an abstracted measurement.

EnviroTrack [1] is an object-based and data-centric middleware designed specifically for embedded tracking applications. EnviroTrack associates a context-label with each entity. Upon initial detection of the entity, the context-label is dynamically created and logically follows the entity's movement through the sensor field. Application developers directly interact with the context label instead of a continuously changing collection of nodes that detect the entity, through the help of a directory service based on a geographic hash table. EnviroTrack relies on embedded sensors with precise knowledge of their locations to locate and track mobile objects.

Finally, Kairos [19] is a macroprogramming model that allows the specification of the network's global behavior through a centralized model. As such, it is not adaptive or general-purpose, requiring deployment-time knowledge of the intended application(s). Regiment [48] also employs a macroprogramming approach to program sensor networks. A user writes a single program which is then distributed and run across the sensor network. Kairos provides abstractions to facilitate this task, while Regiment focuses on the suitability of functional programming to the sensor network domain.

In summary, while these systems for ubiquitous computing have addressed components of the problems associated with the operating environment described in Chapter 1, other components of the problem definition are not completely satisfied by these existing systems. In the next section, we introduce a new paradigm for immersive sensor networks targeted directly towards the application of such networks to the challenges posed by ubiquitous computing applications and their operating environments.

5.2 The DAIS Middleware

In this section, we discuss the high-level middleware model [27, 32] that encapsulates the previously outlined research issues and provides a cohesive environment for ubiquitous computing application development. Figure 5.1 depicts the DAIS [1] (Declarative Applications in Immersive Sensor networks) middleware model. This architecture consists of a handheld component (running on, for example, a laptop or a PDA) and the immersive sensor environment (defined by a community of sensors). The figure shows the explicit hierarchical model of the middleware, which enables more powerful devices (i.e., client devices) to support more of the system's functionality than resource constrained devices (i.e., physical sensors). As Figure 5.1 shows, a client application runs with the support of both the scene abstraction and the virtual sensors abstraction, in addition to several other components.

The types of queries enabled on a scene can be classified into *one-time queries* (which return a single result from each scene member) and *persistent queries* (which return periodic results from scene members). To support these types, we provide two different methods for posing queries to the network. We also include versions of these two methods that request processing of the retrieved data before the result is handed back to the application. Furthermore, a virtual sensor could be created in a scene for abstracting the necessary data.

The remainder of this section describes communication protocol encapsulation envisioned for the middleware, how queries come from the application, and what

[1]DAIS (dā′ĭs): from the middle English word meaning "raised platform"

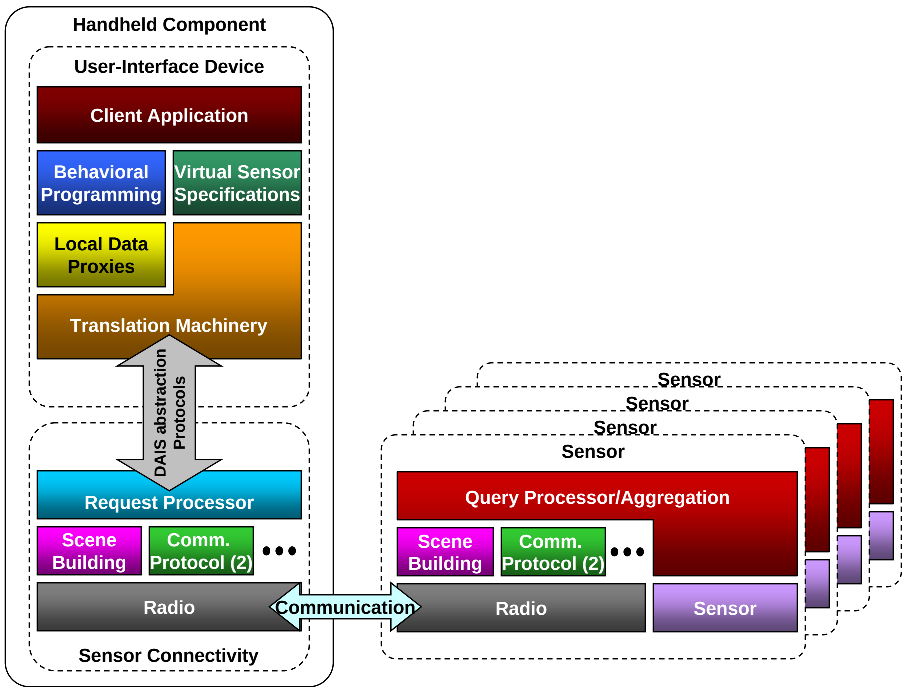


Figure 5.1: The high-level middleware architecture. The left-hand side shows the components comprising the model on the component carried by the user (e.g., PDA or laptop), and the right-hand side shows the middleware components on the sensors.

happens as they travel through the middleware in Figure 5.1. Figure 5.2 depicts a simplified object diagram of the DAIS middleware layers. The names of the layers on the right of the figure correspond to the layers in Figure 5.1.

5.2.1 Communication Protocol Encapsulation

As shown in Figure 5.2, our middleware makes use of the *strategy pattern* [15], a software design pattern in which algorithms (such as strategies for query dissemination) can be chosen at runtime depending on system conditions. The strategy pattern provides a means to define a family of algorithms, encapsulate each one as an object, and make them interchangeable. Such an approach allows the algorithms to vary independently from clients that use them.

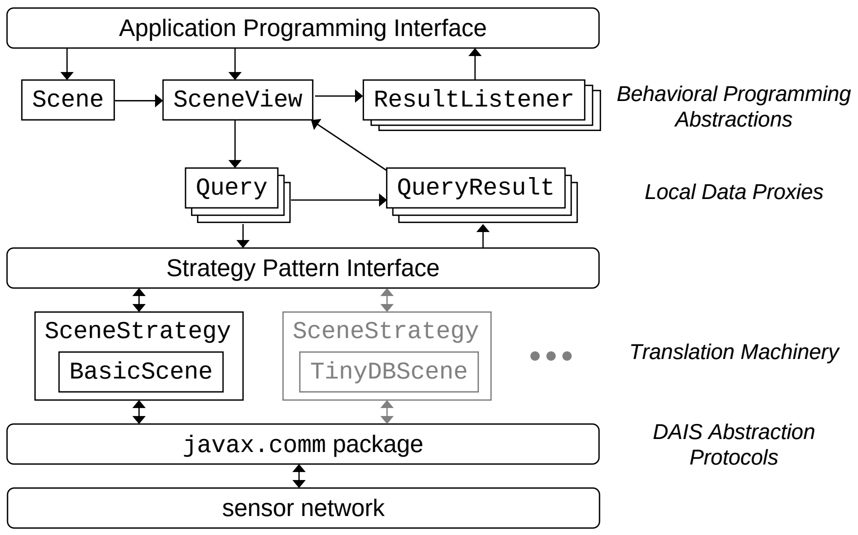


Figure 5.2: Simplified object diagram for DAIS

In DAIS, the clients that employ the strategies are the queries, and the different strategies are the `SceneStrategy` algorithms. These algorithms determine how a `Query` is disseminated to the `Scene` and how the `QueryResult` is returned. If a particular dissemination algorithm other than the default is required for a specific application, an appropriate `SceneStrategy` algorithm is instantiated.

Two principle directives of object-oriented design are used in the strategy pattern: encapsulate the concept that varies, and program to an interface, not to an implementation. Using the strategy pattern, we decouple the `Query` from the code that runs it so we can vary the query dissemination algorithm without modifying the `Query` class. The loose coupling that the strategy pattern enables between the components makes the system easier to maintain, extend, and reuse.

The `BasicScene` refers to an implementation of a naive version of the scene abstraction from Chapter 2, in which all data aggregation is performed locally. Other communication approaches can be swapped in for the `BasicScene` (for example one built around TinyDB [44] or directed diffusion [25], although the implementations of these approaches on the sensors may have to be modified slightly to accommodate scene construction). By defining the `SceneStrategy` interface, we enable developers who are experts in existing communication approaches to create simple plugins that use different query dissemination and/or aggregation protocols. Different communication paradigms can be used in different environments or to support different application domains depending on the resource constraints or domain-specific capabilities of the devices in a particular domain.

Each `SceneStrategy` interacts with the `javax.comm` package to provide the *DAIS abstraction protocols* that allow the portion of the middleware implemented in Java (described above) to interact with the sensor hardware. Each `SceneStrategy` requires not only a high-level portion implemented on the handheld device, but also a low-level portion that runs on the sensors. In the next section, we detail the DAIS middleware, through an example (that uses the `BasicScene` strategy).

5.2.2 Processing Dynamic Queries: A Step-By-Step Example

To describe the model of DAIS, we follow a query from the application developer's hands all the way into the network to the designated sensor and back. To present the steps involved in the process, we use a single, specific application example taken from the intelligent construction site domain. In the example we have selected, the application's user would like to receive a reading of any crane load within 100m that exceeds 15 tons. The user would like this information to be updated every 10 seconds to ensure he has the most current information.

Step 1: Declare a Scene. This first step is performed by the application developer through the programming interface. Nothing happens involving network communication until the application actually uses the scene. This is beneficial in terms of reducing communication overhead. For our application example, the developer uses the following code to invoke the constructor of a `Scene` object that defines a scene that includes every sensor (not just those measuring crane weight) within 100m of the declaring device:

```
Scene s = new Scene({ new Constraint( Scene.SCENE_DISTANCE,
                                      Scene.SCENE_DFORMULA,
                                      new IntegerThreshold(100)) });
```

When the application subsequently needs to query the constructed Scene, it calls the `getSceneView()` method which returns a handle to a `SceneView`. The application can then use this `SceneView` to send a `Query` over the `Scene`.

Step 2: Issue a query. This step is performed by the application developer using the `SceneView` instance created and accessed in the previous step. In our example application, the developer must first create the query:

```
Query q = new Query( new Constraint(''Equipment'',
                                    Query.EQUALS_OPERATOR,
                                    ''Crane''),
                     new Constraint(''Weight'',
                                    Query.GT_OPERATOR,
                                    15)} );
```

In this case, the `Query` is defined by two `Constraints`. The first requires the sensor used to belong to a piece of equipment that has the label "Crane." This prevents the query from discovering weight sensors on, for example, dump trucks. The second constraint limits the sensors that respond to the query to only those that measure a load with a weight of more than 15 tons. As discussed in the subsequent steps below, *every* sensor receiving this query (i.e., all sensors within 100m) that have weight sensors periodically evaluate the query, but only respond if and when the load sensed exceeds 15 tons. After creating the `Query` above and a `ResultListener, r` to receive the results (omitted for brevity), the application developer dispatches it using the `SceneView`:

```
SceneView sv = s.getSceneView();
int receipt = sv.registerQuery(q, r, 10);
```

where 10 refers to the period with which the application wishes to sense crane load weights. Upon receiving any query request, the `SceneView` object adjusts its state in several ways. First, for every query, a table within the `SceneView` is updated with a mapping from a unique query id generated for the query to the `ResultListener` handle provided with the query. In addition, for persistent queries, this unique id is returned as a receipt of registration that can be used in subsequent interactions to deregister the query.

Step 3: Local Data Proxy is Created. From this point on, control passes from the application developer to the middleware which is now responsible for ensuring that the application's `ResultListener` is called with the appropriate data at the appropriate times. The first step requires DAIS to create a local proxy for this query to handle return calls for this query. The local data proxy is especially important in facilitating the translation between the low-level language spoken by the sensors in the network and the high-level language the application uses.

Step 4: Construct and Distribute Protocol Query. The local data proxy within the DAIS middleware transforms the application's request into a protocol data unit for the scene implementation in use. As shown in Figure 5.2, several different protocols can provide the communication functionality as long as they adhere to the specified strategy pattern interface. The scene implementations handle both persistent and one-time queries. In our current implementation of a scene communication protocol [28], the scene protocol message carries the information about scene membership constraints *and* the data query at the same time. This reduces the communication overhead by constructing the scene on-demand. The details of the communication protocol are omitted here; it suffices to say that, by its definition, the protocol ensures that the data query is delivered to the set of sensor nodes that satisfy the scene's constraints. In our example application, this means that every sensor within 100m will receive the data query constructed above.

Step 5: Scene Query Processed by Remote Sensor. We use TinyOS to implement functionality on the sensors; when the communication protocol receives and processes a scene message, if it determines that the node is within the scene, it passes the received message up to the application layer. In DAIS, the application layer is defined by the `QueryProcessor`, again, implemented in TinyOS. This is essentially a slimmed down version of our middleware that is capable of running on the resource constrained device. An abstract representation of the TinyOS implementation of the `QueryProcessor` is shown in Figure 5.3. In TinyOS jargon, the picture shows *components* as rounded rectangles and *interfaces* as arrows connecting components. A component *provides* an interface if the corresponding arrow points toward it and *uses* an interface if the arrow points away from it. If a component provides an interface, it must implement all of the *commands* in the interface, and if the component uses an interface, it can call any commands in the interface and must handle all *events* generated by the interface.

In our implementation, the `QueryProcessor` component provides the functionality shown at the top layer of the sensor portion of the architecture in Figure 5.1. The query arrives in the `QueryProcessor` through the `receive` event of the `Receive` interface. If the query is a one-time query (as indicated by a field in the TinyOS message), then the `QueryProcessor` simply connects to the on-board sensor that can provide the requested data type (depicted as `Sensor` in the figure) through the `ADC` interface. If the data request is for a sensor type that is not supported on this device (i.e., the sensor table stored in the

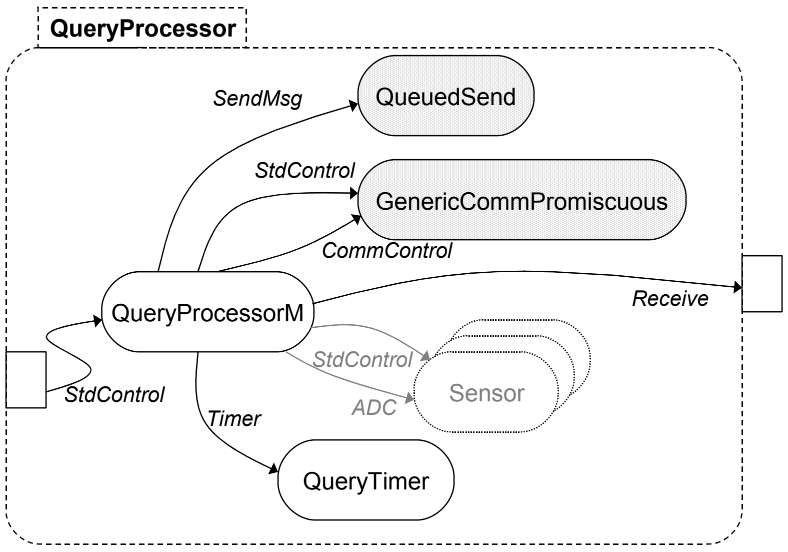


Figure 5.3: Implementation of the `QueryProcessor` functionality on sensors

`QueryProcessor` has no mapping to a sensor that can provide the specified data type), then the message is ignored. The node is still included in the scene because it may be a necessary routing node connecting the requester to another node that *does* have the required sensor.

If the query is persistent (as in our crane load example), then in addition to immediately returning the requested value, the `QueryProcessorM` module also initializes a `QueryTimer` using the request frequency specified in the `data` portion of the received message. When the timer fires, `QueryProcessorM` retrieves a value from the sensor and sends it back to the initial requester using the sendMsg interface of the `QueuedSend` module.

When a node is no longer in a scene, the scene communication implementation creates a null message that it sends to the `QueryProcessor` through the Receive interface. The `QueryProcessor` takes this message as a sign to cease streaming data back to the requester, and stops the `QueryTimer`.

Step 6: Query Processor Replies. If the query processor possessed the correct sensor *and* the other components of the query are also satisfied (e.g., the constraints on the data value), the query processor replies (and continues to reply periodically to a persistent query. In our example application, a sensor node's `QueryProcessor` will reply every 10 seconds if the sensor is attached to a crane, it has a weight sensor, and the value from the sensor is greater than 15 tons. If the weight is not originally greater than 15 tons, the sensor does not reply unless the weight becomes greater than 15 tons. This reply is sent through the `SendMsg` interface shown in Figure 5.3 and uses basic multihop routing to return to the original requester.

Step 7: Result Received by Client Device. After propagating through the underlying communication substrate, query replies will arrive at the client device's sensor network interface. At the client device, the result is demultiplexed by the *Result Processor* (shown in Figure 5.1) and handed off to the appropriate local proxy. Again, the local proxy is automatically generated and managed by the middleware. It translates the low-level query reply to a high-level `Result` and invokes the application's registered `ResultListener`'s `resultReceived` method. It is important to note that, as shown in Figure 5.2, multiple queries may be active over a single scene at any given time. For each scene, the `SceneView` controls all of these queries and connects them to the underlying implementation via the *strategy pattern interface.* At this point, control transfers back to the application and its `ResultListener` which handles the query's result (or queries' results if multiple matches existed). For persistent queries, as more results arrive, the same process occurs until the application deregisters the query.

5.3 Chapter Summary

In this chapter, we first gave an overview existing middleware systems. We then realized the conceptual model from Chapter 2 and Chapter 3 in a middleware implementation. We also discussed the details of the implementation of the middleware, and gave an example of its use by describing how the middleware handles the application in steps. DAIS is a tiered middleware that allows developers to create lightweight applications that run on client devices (e.g., laptops or PDAs) and interact directly with an immersive sensor environment. Through the virtual sensor and scene abstractions, application developers can declaratively specify the components over which their interactions occur. These abstractions are necessary to simplifying the application development task for ubiquitous computing environments, which are beginning to demand rapid development and deployment of applications in widely varying domains. By allowing different portions of the middleware to be deployed on devices with differing capabilities, DAIS lends itself naturally to the mixed ubiquitous computing environment. The abstractions present in the DAIS middleware provide intuitive and easy-to-use wrappers for complex underlying interactive behaviors. In summary, DAIS presents a unique view of programming pervasive computing environments that in the future will include large numbers of heterogeneous wireless sensors. By creating high-level programming abstractions that encapsulate the locality of pervasive computing interactions, DAIS is a first step in enabling novice programmers to create sophisticated pervasive computing applications.

Chapter 6

Conclusion

This dissertation reports on an effort to simplify application development for programmers of sensor networks. The protocols currently available to support communication and coordination on lightweight, resource-constrained sensors are tailored to application situations, in which sensor networks are commonly accessed through a central collection point. The need to support emerging ubiquitous computing applications calls for a reexploration of protocol and coordination issues targeted for immersive computing environments. In this dissertation, we created and evaluated basic protocols, formalized a programming model for the specification of scenes by novice developers, formally defined virtual sensors and their programming model, as well as remote deployment of virtual sensors. We created a coherent middleware for ubiquitous computing that encapsulates both the virtual sensor and the scene and provides an integrated high-level programming interface.

We presented the scene data communication abstraction, a new communication paradigm tailored to immersive sensor networks that support pervasive computing. Specifically, the scene abstraction is the first to support dynamic client devices roving among sensors. The scene abstraction and the protocol that implements it utilize a high degree of context-awareness and adaptation. This allows an application's scene to consistently reflect its instantaneous operating environment. Using the scene abstraction and protocol, an application has a direct view of the information sources available in the immediate environment. In addition, our approach combines this local perspective with a communication protocol for disseminating client requests and returning replies from the scene participants. We presented the abstraction, its implementation, and an initial feasibility study of its performance. Future work will include a more complete evaluation to include measurements of the approach's expressiveness through a larger-scale real-world deployment on a mixture of embedded and client devices.

We also defined a new virtual sensor model designed to abstract data from heterogeneous physical sensors by applying user-defined functions. The separation of the specification of the sensing task from the sensing behavior allows a programmer to describe the behavior of a virtual sensor, without having to specify the underlying details of how it should be constructed. We realized the model in a middleware implementation for the creation of virtual sensors enabling adaptive and efficient in-network processing that dynamically responds to an application's needs. This implementation was demonstrated to support applications in two different domains. Virtual sensors offer a way to tailor a generic sensing environment to specific applications. This will be especially necessary as sensor networks become more widespread and general-purpose.

Several directions can be envisioned that build directly on the work presented in this dissertation. With respect to scenes, an interesting point of discussion is that of the degree of adaptivity the scene abstraction provides. We motivated throughout the dissertation that awareness of and adaptation to the surrounding environment are crucial to enabling pervasive computing applications. The scene abstraction as described in this dissertation already incorporates several points of adaptation, most specifically allowing the participants in a scene to change over time in response to client mobility or to changes in the network or physical environment. We discussed such adaptation with respect to setting the scene's beacon frequency to be sensitive to the client mobility or a sensor node's local perception of mobility. Future work could explore additional adaptation points that could make the abstraction even more responsive to pervasive computing applications. For instance, one could imagine a scene's threshold expanding or contracting based on the environmental values sensed or the density of available readings.

With respect to remotely deploying virtual sensors, the optimality of the two different approaches in different application situations could be further explored. Another area for future work is to design the ability for a remotely deployed virtual sensor to intelligently follow its creator's device as it moves through the sensor network. On a construction site, a supervisor may deploy a virtual sensor to monitor the movement of the nearest crane. As the supervisor moves through the site, this will require the remotely deployed virtual sensor to remain aware of the user or application's relative location and to adjust its location accordingly. Furthermore, the input data types could be extended to carry more semantics than simply the nature of the physical measure provided by a sensor. They could carry the physical data type (e.g., "temperature") and location context information (e.g., "on top of", "under", etc.) to provide some sense of the relationship between these physical temperature readings and the abstract measure we are trying to evaluate.

Other future work could focus on automatically generating virtual sensors through the addition of simple functions in the ontology. A question that arises in the context of a virtual sensor and physical sensor that can provide the same data type is "If the user is outdoors and both triangulation and GPS are available, which one should be used?" To make this decision on behalf of the user, some cost metrics can be added to the ontology. Future work could also explore supporting more complicated interactions such as what to do when high frequency queries are combined with high frequency update rates. Moreover, various physical sensors may have different update frequencies (and costs associated with obtaining and relaying these updates), so the decision should depend on the total expected costs (calculated most likely using some statistical properties).

Bibliography

[1] T. Abdelzaher, B. Blum, Q. Cao, Y. Chen, D. Evans, J. George, S. George, L. Gu, T. He, S. Krishnamurthy, L. Luo, S. Son, J. Stankovic, R. Stoleru, and A. Wood. EnviroTrack: Towards an environmental computing paradigm for distributed sensor networks. In *Proc. of ICDCS*, pages 582–589, 2004.

[2] K. Aberer, M. Hauswirth, and A. Salehi. Middleware support for the "internet of things". In *GI/ITG KuVS Fachgespräch "Drahtlose Sensornetze" (Expert Talk on Wireless Sensor Networks), Universität Stuttgart*, 2004.

[3] A. Bakshi, V. K. Prasanna, J. Reich, and D. Larner. The abstract task graph: A methodology for architecture-independent programming of networked sensor systems. In *Proc. of the Wkshp. on End-to-end Sense-and-respond Sys.*, pages 19–24, 2005.

[4] B. Blum, P. Nagaraddi, A. Wood, T. Abdelzaher, S. Son, and J. Stankovic. An entity maintenance and connection service for sensor networks. In *Proc. of MobiSys*, pages 201–214, 2003.

[5] A. Boulis, C.-C. Han, and M. Srivastava. Design and implementation of a framework for efficient and programmable sensor networks. In *Proceedings of the 2nd International Conference on Mobile Systems, Applications, and Services*, pages 187–200, 2003.

[6] M. Brown, S. Gilbert, N. Lynch, C. Newport, T. Nolte, and M. Spindel. The virtual node layer: a programming abstraction for wireless sensor networks. *SIGBED Rev. Special issue on the workshop on wireless sensor network architecture*, 4(3):7–12, April 2007.

[7] E. Cheong, J. Liebman, J. Liu, and F. Zhao. TinyGALS: A programming model for event-driven embedded systems. In *Proc. of the 2003 ACM Symposium on Applied Computing*, pages 698–704, 2003.

[8] P. Ciciriello, L. Mottola, and G. P. Picco. Building virtual sensors and actuators over logical neighborhoods. In *Proc. of the 1st ACM International Workshop on Middleware for Sensor Networks*, 2006.

[9] P. Corsini, P. Masci, and A. Vecchio. VirtuS: A configurable layer for post-deployment adaptation of sensor networks. In *Proc. of the Int'l Conf. on Wireless and Mobile Comm.*, pages 8–13, 2006.

[10] P. Costa, L. Mottola, A. Murphy, and G.P. Picco. TeenyLIME: Transiently shared tuple space middleware for wireless sensor networks. In *Proc. of the International Workshop on Middleware for Sensor Networks*, pages 43–48, 2006.

[11] Crossbow Technology: Wireless Sensor Networks: Cricket. http://www.xbow.com/Products/productdetails.aspx?sid=176, 2008.

[12] Crossbow Technologies, Inc. http://www.xbow.com, 2008.

[13] C. Curino, M. Giani, M. Giorgetta, A. Giusti, A. Murphy, and G.P. Picco. TinyLIME: Bridging mobile and sensor networks through middleware. In *Proc. of the 3rd Int'l. Conf. on Pervasive Computing and Communications*, pages 61–72, 2005.

[14] C.-L. Fok, G.-C. Roman, and C. Lu. Rapid development and flexible deployment of adaptive wireless sensor network applications. In *Proc. of the 25th Int'l. Conf. on Distributed Computing Systems*, pages 653–662, 2005.

[15] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns.* Addison-Wesley, 1995.

[16] D. Gay, P. Levis, R. vonBehren, M. Welsh, E. Brewer, and D. Culler. The nesC language: A holistic approach to networked embedded systems. In *Proc. of the ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 1–11, 2003.

[17] L. Girod, J. Elson, A. Cerpa, T. Stathopoulous, N. Ramanathan, and D. Estrin. EmStar: A software environment for developing and deploying wireless sensor networks. In *Proc. of the 2004 USENIX Technical Conference*, pages 283–296, 2004.

[18] B. Greenstein, E. Kohler, and D. Estrin. A sensor network application construction kit (SNACK). In *Proc. of the 2nd Int'l. Conf. on Embedded Networked Sensor Systems*, pages 69–80, 2004.

[19] R. Gummadi, O. Gnawali, and R. Govindan. Macro-programming wireless sensor networks using kairos. In *Proc. of the Int'l. Conf. on Dist. Comp. in Sensor Sys.*, pages 126–140, 2005.

[20] J. Hammer, C. Julien, S. Kabadayi, W. O'Brien, and J. Trujillo. Dynamic decision support in direct-access sensor networks: A demonstration. In *Proc. of the 3rd Int'l Conf. on Mobile Ad Hoc and Sensor Systems*, pages 578–581, 2006.

[21] W. Heinzelman, A. Muprhy, H. Carvallo, and M. Perillo. Middleware to support

sensor network applications. *IEEE Network Magazine Special Issue*, 18(1):6–14, 2004.

[22] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister. System architecture directions for networked sensors. In *Proc. of the 9th Int'l. Conf. on Architectural Support for Programming Languages and Operating Systems*, pages 93–104, 2000.

[23] Q. Huang, C. Lu, and G.-C. Roman. Spatiotemporal multicast in sensor networks. In *Proc. of the 1st Int'l. Conf. on Embedded Networked Sensor Systems*, pages 205–217, 2003.

[24] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In *Proc. of the 2nd Int'l. Conf. on Embedded Networked Sensor Systems*, pages 81–94, 2004.

[25] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heideman, and F. Silva. Directed diffusion for wireless sensor networking. *IEEE/ACM Transactions on Networking*, 11(1):2–16, February 2003.

[26] D. Johnson, D. Maltz, and J. Broch. DSR: The dynamic source routing protocol for multi-hop wireless ad hoc networks. *Ad Hoc Networking*, pages 139–172, 2001.

[27] C. Julien and S. Kabadayi. Enabling programmable ubiquitous computing environments: a middleware perspective. In S. K. Mostéfaoui, Z. Maamar, and G. Giaglis, editors, *Advances in Ubiquitous Computing: Future Paradigms and Directions*, chapter 5, pages 117–149. IGI Publishing, Hershey, PA, 2008.

[28] S. Kabadayi and C. Julien. A local data abstraction and communication paradigm for pervasive computing. In *Proc. of the 5th Annual IEEE Interna-*

tional Conference on Pervasive Computing and Communications, pages 57–68, 2007.

[29] S. Kabadayi and C. Julien. Remotely deployed virtual sensors. In *Technical Report TR-UTEDGE-2007-010*, 2007.

[30] S. Kabadayi and C. Julien. Scenes: abstracting interaction in immersive sensor networks. *Elsevier Pervasive and Mobile Computing: Special Issue on Selected Papers from PerCom 2007*, 3(6):635–658, December 2007.

[31] S. Kabadayi, C. Julien, W.J. O'Brien, and D. Stovall. Virtual sensors: A demonstration. In *26th Int'l Conf. on Computer Communications: Demonstrations Track*, 2007.

[32] S. Kabadayi, C. Julien, and A. Pridgen. DAIS: Enabling declarative applications in immersive sensor networks. In *Technical Report TR-UTEDGE-2006-000*, 2006.

[33] S. Kabadayi, C. Julien, and J. Trujillo. Virtual sensors: Heterogeneous aggregation in pervasive networks. Technical Report TR-UTEDGE-2006-009, The Center for Excellence in Distributed Global Environments, The University of Texas at Austin, 2006.

[34] S. Kabadayi, A. Pridgen, and C. Julien. Virtual sensors: Abstracting data from physical sensors. In *Proc. of the 4th International Workshop on Mobile Distributed Computing*, pages 587–592, 2006.

[35] C. Kidd, R. Orr, G. Abowd, C. Atkeson, I. Essa, B. MacIntyre, E. Mynatt, T. Starner, and W. Newstetter. The aware home: A living laboratory for ubiquitous computing research. In *Proc. of the 2nd Int'l. Workshop on Cooperating Buildings, Integrating Information, Organization and Architecture*, pages 191–198, 1999.

[36] J. Kosh and R. Pandey. VM*: Synthesizing scalable runtime environments for sensor networks. In *Proc. of the 3rd ACM Conf. on Embedded Networked Sensor Systems*, 2005.

[37] Lego Store - Building Crane. http://shop.lego.com/Product/?p=7905, 2008.

[38] P. Levis and D. Culler. Maté: A tiny virtual machine for sensor networks. In *Proc. of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 85–95, 2002.

[39] P. Levis, N. Lee, M. Welsh, and D. Culler. TOSSIM: Accurate and scalable simulation of entire tinyos applications. In *Proc. of SenSys*, 2003.

[40] J. Liu, M. Chu, J. Reich, J.J Liu, and F. Zhao. State-centric programming for sensor-actuator network systems. *IEEE Pervasive Computing*, 2(4):50–62, October-December 2003.

[41] C. Lu, G. Xing, O. Chipara, C.-L. Fok, and S. Bhattacharya. A spatiotemporal query service for mobile users in sensor networks. In *Proc. of the 25th Int'l. Conf. on Distributed Computing Systems*, pages 381–390, 2005.

[42] L. Luo, T. Abdelzaher, T. He, and J. Stankovic. EnviroSuite: an environmentally immersive programming framework for sensor networks. *ACM Transactions on Embedded Computing Systems*, 5(3):543–576, August 2006.

[43] S. Madden, M. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad hoc sensor networks. In *Proc. of the 5th Symp. on Operating Systems Design and Implementation*, pages 131–146, 2002.

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

[45] Crossbow Technology: Wireless Sensor Networks: MICA2. http://www.xbow.com/Products/productdetails.aspx?sid=174, 2008.

[46] L. Mottola and G.P. Picco. Programming wireless sensor networks with logical neighborhoods. In *Proc. of InterSense*, 2006.

[47] L. Neitzel, S. Seixas, and K. Ren. A review of crane safety in the construction industry. *Applied Occupational and Environmental Hygiene*, 16(12):1106–1117, 2001.

[48] R. Newton and M. Welsh. Region streams: functional macroprogramming for sensor networks. In *Proc. of the 1st Int'l' Workshop on Data Management for Sensor Networks*, pages 78–87, 2004.

[49] Y. Ni, U. Kremer, A. Stere, and L. Iftode. Programming ad-hoc networks of mobile and resource-constrained devices. In *Proc. of PLDI*, pages 249–260, 2005.

[50] V. Rajamani, S. Kabadayi, and C. Julien. Query domains: Grouping heterogeneous sensors based on proximity. In *Proceedings of the 3rd IEEE International Workshop on Heterogeneous Multi-Hop Wireless and Mobile Networks*, 2007.

[51] Remotely Deployed Virtual Sensors Implementation Code. http://mpc.ece.utexas.edu/remotevirtualsensors/index.html, 2007.

[52] G.-C. Roman, C. Julien, and Q. Huang. Network abstractions for context-aware mobile computing. In *Proc. in the 24th Int'l. Conf. on Software Engineering*, pages 363–373, 2002.

[53] M. Roman, C. Hess, R. Cerqueira, A. Ranganathan, R. Campbell, and K. Narstedt. A middleware infrastructure for active spaces. *IEEE Pervasive Computing*, 1(4):74–83, 2002.

[54] RPN Calculator. http://home.att.net/~srschmitt/script_reverse_polish.html, 2008.

[55] N. Shrivastava, C. Burgohain, D. Agrawal, and S. Suri. Medians and beyond: New aggregation techniques for sensor networks. In *Proc. of the 2nd Int'l. Conf. on Embedded Networked Sensor Systems*, pages 239–249, 2004.

[56] S. Voida, E. Mynatt, and B. MacIntyre. Supporting collaboration in a context-aware office computing environment. In *Proc. of the Workshop on Collaboration with Interactive Walls and Tables*, 2002.

[57] Virtual Sensors. http://mpc.ece.utexas.edu/virtualsensors/index.html, 2006.

[58] M. Welsh and G. Mainland. Programming sensor networks using abstract regions. In *Proc. of the 1st USENIX/ACM Symp. on Networked Systems Design and Implementation*, 2004.

[59] K. Whitehouse, C. Sharp, E. Brewer, and D. Culler. Hood: A neighborhood abstraction for sensor networks. In *Proc. of the 2nd Int'l. Conf. on Mobile Systems, Applications, and Services*, pages 99–110, 2004.

[60] K. Whitehouse, F. Zhao, and J. Liu. Semantic streams: A framework for composable semantic interpretation of sensor data. In *Proc. of EWSN*, pages 5–20, 2006.

[61] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *ACM SIGMOD Record*, 31(3):9–18, 2002.

[62] Y. Yu, B. Krishnamachari, and V.K. Pasanna. Issues in designing middleware for wireless sensor networks. *IEEE Network*, 18(1):15–21, 2004.

[63] Y. Yu, L. J. Rittle, V. Bhandari, and J. B. LeBrun. Supporting concurrent

applications in wireless sensor networks. In *Proc. of the Fourth Int. Conf. on Embedded Networked Sensor Systems*, pages 139–152, 2006.

Vita

Sanem Kabadayi received her B.S. in Electrical Engineering and B.S. in Physics degrees from the University of Texas at Austin in 2000. She received her M.S.E.E.C.S. from SabancıUniversity in 2002. While a Ph.D. student in the Department of Electrical and Computer Engineering at the University of Texas at Austin, she was a member of the Mobile and Pervasive Computing Group. Her current research interests include wireless networking, sensor networks, and pervasive computing.

Permanent Address: 3487 Lake Austin Blvd Apt C, Austin, TX 78703

This dissertation was typeset with LaTeX 2_ε[1] by the author.

[1]LaTeX 2_ε is an extension of LaTeX. LaTeX is a collection of macros for TeX. TeX is a trademark of the American Mathematical Society. The macros used in formatting this dissertation were written by Dinesh Das, Department of Computer Sciences, The University of Texas at Austin, and extended by Bert Kay, James A. Bednar, and Ayman El-Khashab.