DAIS: Enabling Declarative Applications in Immersive Sensor Networks

Sanem Kabadayi
Christine Julien
Adam Pridgen

TR-UTEDGE-2006-000

DAIS: Enabling Declarative Applications in Immersive Sensor Networks

Sanem Kabadayi, Christine Julien, and Adam Pridgen
The Center for Excellence in Distributed Global Environments
The Department of Electrical and Computer Engineering
The University of Texas at Austin
{s.kabadayi, c.julien, atpridgen}@mail.utexas.edu

Abstract

As sensor networks become increasingly ubiquitous, we envision an instrumented environment that can provide varying amounts of information to mobile applications immersed within the network. Such a scenario deviates 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. Instead, we target future scenarios in which multiple mobile applications will leverage sensor network nodes opportunistically and unpredictably. This paper introduces the DAIS (Declarative Applications in Immersive Sensor networks) middleware platform that enables the development of these adaptive mobile applications. Our approach focuses on minimizing communication in the sensor network to best ensure the network's lifetime. DAIS localizes 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 DAIS accomplishes this through a novel programming abstraction that enables on-demand access to dynamic data sources. This paper reports on our initial work with the DAIS middleware platform and highlights both the programming interface and its interaction with the necessary underlying communication constructs.

Keywords: sensor networks, middleware, context-awareness, coordination, declarative languages

1 Introduction

Sensor networks have emerged as an integral component of pervasive computing environments and are rapidly becoming ubiquitous. Such networks consist of numerous miniature, battery-powered devices that communicate wirelessly to gather and share information about the instrumented environment. While many new concerns arise in comparison to existing distributed or mobile computing scenarios, potential applications of this technology abound and range from intrusion detection to habitat monitoring.

To date, much application development has been limited to academic circles. One significant barrier to the widespread development of these sensor network applications lies in the increased complexity of the programming task when compared to existing distributed or even mobile situations. Sensor nodes are severely resource-constrained, in terms of both computational capabilities and battery power, and therefore an application development task must inherently consider low-level design concerns. This complexity, coupled with the increasing demand for sensor applications, highlights the need for programming platforms (i.e., *middleware*) that simplify application development.

As described in more detail in later sections, much of the existing work in simplifying programming in sensor networks focuses on application-specific networks where the nodes are statically deployed for a particular task. We envision a more futuristic (but not unrealistic) scenario in which sensor networks become more general-purpose and reusable. While the networks may remain domain-specific, we target situations in which the applications that will be deployed are not known *a priori* and may include varying sensing needs and adaptive behaviors. Examples of such domains include aware homes [16], intelligent construction sites [15], battlefield scenarios [12], and first responder deployments [18]. Finally, existing applications commonly assume that sensor data is collected at a central location to be processed and used in the future and/or accessed via the Internet. Applications from the domains described above, however, involve users immersed in the sensor network who access locally sensed information on-demand. This is exactly the vision of future pervasive computing environments [27], in which sensor networks must play an integral role [4].

This paper introduces the DAIS [1] (Declarative Applications in Immersive Sensor networks) middleware platform that provides programming abstractions tailored to the ubiquitous applications described above. DAIS is not a middleware for sensor networks in the sense that it runs strictly on the sensors. Instead, DAIS allows developers to create applications that run on *client devices* (e.g., laptops or PDAs) that interact directly with embedded sensor networks.

While this style of interaction is common in many application domains, throughout the paper we will use applications from an *intelligent job site* to motivate the middleware and its provided constructs. Such an environment provides a unique and heterogeneous mix of sensing and mobile devices. The former includes sensors on equipment to measure its location, within concrete to measure temperature and generate cured data, on cranes to measure their movement and stresses, and even near hazardous materials to measure the concentration of potentially dangerous chemicals [15]. Mobile devices include those moving within vehicles or carried by workers. The applications in this scenario demand opportunistic interaction with locally available sensors, and we will use them to demonstrate the use of DAIS.

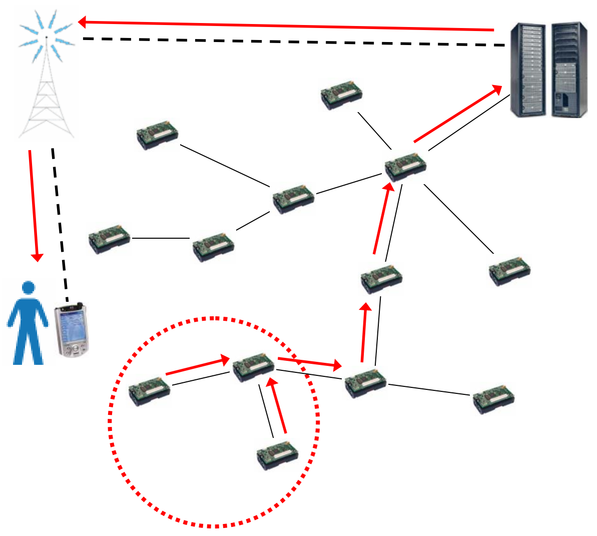
The specific novel contributions of this work are threefold. First, we describe a new middleware architecture designed to mediate direct interactions between mobile devices and an immersive sensor network. Second, we create an essential underlying abstraction that allows a developer to precisely specify the dynamic set of sensors that it interacts with. Finally, we describe a prototype implementation of this middleware that includes the creation of a protocol enabling adaptive and efficient construction of local zones within the sensor network that dynamically reflect an application's needs.

Section 2 of this paper describes the challenges, highlighting how immersive sensor environments change the requirements of a middleware. In Section 3, we provide a detailed description of the DAIS architecture. Section 4 covers the internals of the middleware. In Section 5, we evaluate a case study in scene definition. Section 6 contrasts related projects, and Section 7 concludes.

2 Problem Definition

In *immersive sensor networks*, users move through an instrumented environment and desire to have on-demand access to information gathered from the local area. Figure 1(a) shows how current interactions with sensor networks commonly occur, while Figure 1(b) shows the needs of our intended applications. While existing collection behaviors that sense, aggregate, and stream information to a central collection point may be useful in immersive sensor networks, this paper undertakes only the portion of the problem related to enabling applications' direct, on-demand interactions. From this perspective, our approach focuses on using sensor networks to enable pervasive computing applications, not on remote distributed sensing.

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


(a) Using today's capabilities, a user's query is physically detached from the sensor field. Resolution of the query relies on a centralized collection point, generally the *root* of a routing tree constructed over the network.

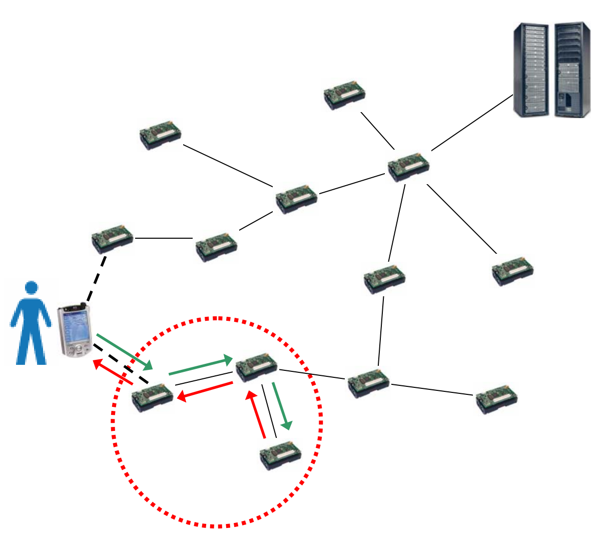

(b) Using DAIS, an application running on the user's device can seamlessly connect to the sensors in the local region, removing the requirement that physically distant sensors participate in the query's resolution.

Figure 1. Comparison of (a) existing operational environments with (b) the operational environment imposed by DAIS.

This style of interaction differs from common uses of sensor networks, introducing several unique challenges and heightening existing ones:

- *Locality of interactions*: An application interacts directly with local sensor nodes. While this can minimize the communication overhead and latency (see Figure 1), 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*: Immersive sensor networks demand 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*: application development for sensor networks is largely recognized as a difficult task. In immersive sensor networks, the programming burdens are magnified due to the challenges described here. 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 for flexible yet expressive programming environments that enable application development for immersive sensor networks while paying careful attention to resource constraints. DAIS addresses these concerns by providing intuitive abstractions of the immersive environment.

3 The DAIS Middleware Model

In this section, we discuss the high-level middleware model used in DAIS. Figure 2 shows the DAIS architecture, which consists of a handheld component (running on either a laptop or a PDA) and the immersive sensor environment (defined by a community of sensors). As Figure 2 shows, a client application

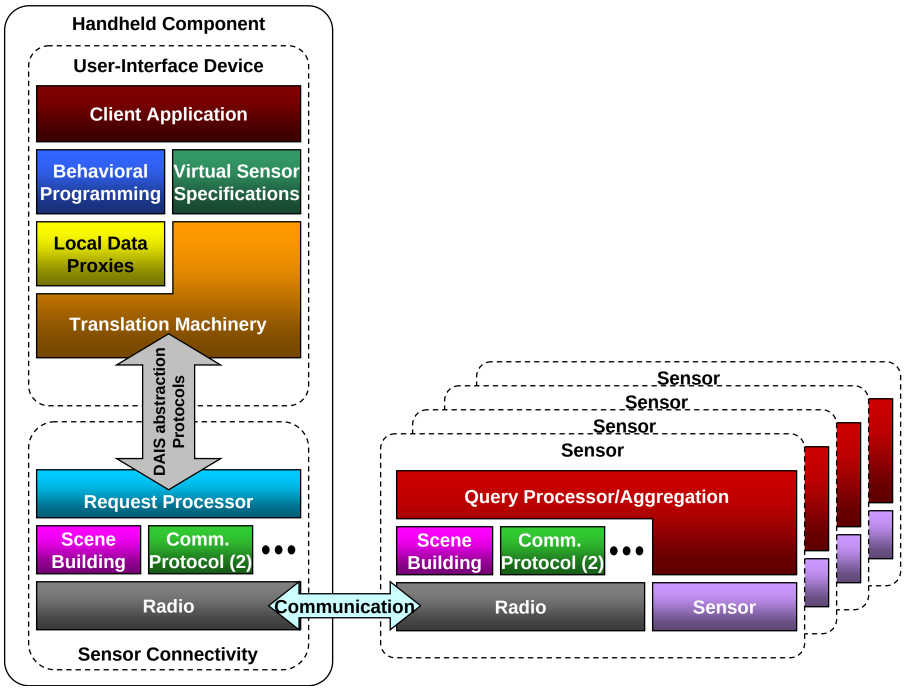


Figure 2. The DAIS high-level architecture. The left-hand side shows the components comprising the model on the component carried by the user (e.g., PDA or laptop), and the right-hand side shows the DAIS components on the sensors.

runs with the support of two key abstractions, namely *behavioral programming abstractions* and *virtual sensors*. Through interfaces provided by these components, the application developer not only defines his applications' interaction environments, but can also programmatically specify their data requests.

The second of these components, virtual sensors, allow applications to specify sophisticated aggregation mechanisms for heterogeneous data sources. Such 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 the cranes to have safe load indicators that determine if a crane is exceeding its capacity. Such a virtual sensor would take measurements from physical sensors that monitor boom angle, load, telescoping length, two-block conditions, wind speed, etc. [22]. Signals from these individual sensors are used in calculations within the virtual sensor to determine if the crane has exceeded its safe working load.

In this paper, we focus not on the definition of virtual sensors, but on the behavioral programming abstractions DAIS provides for application development on top of a set of physical and virtual sensors. We first describe how a developer defines the locality of his applications' interactions using an abstraction called the *scene*. We then detail how programs dynamically interact with data from the environment through an intuitive programming interface. The details of the underlying implementation (i.e., the lower levels of Figure 2 and all of the components on-board the sensors) will be described in Section 4.

3.1 Scenes: Declarative Specifications of Local Interactions

In an instrumented sensor network, a user's operational context is highly dynamic. That is, the set of available data sources changes based on the user's location and movement. Furthermore, if the sensor 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. For an efficient approach, the application must be able to limit the scope of its interactions to include only the data that matches its needs.

In our model, an application's operating environment (i.e., the sensors with which it interacts) is encapsulated in an abstraction called a *scene*. By its definition, a scene constrains which particular sensors may influence an 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. The declarative specification defining a scene allows a programmer to describe the type of scene he wants to create, without requiring him to specify the underlying details of how it should be constructed. The programmer only needs to specify the following three parameters for the scene:

- Metric: A quantifiable property of the network or physical environment that defines the cost of a connection.
- Aggregator: An aggregation function (SUM, AVG, MIN, MAX) that operates on link weights in a network path to calculate the cost of the path.
- Threshold: The threshold value that the cost calculated for a path must satisfy for that sensor to be a member of the scene.

The use of the aggregator to define a scene is different from existing data aggregation schemes in sensor networks [11, 20, 26]. In this case, we are using the same notion of aggregation to calculate the expanse of a scene instead of to consolidate data readings en route back to a requester. We revisit the use of this Aggregator interface for a more traditional purpose in Section 3.2.

Even though the scene concept conveys a notion of locality, an application may choose its scene to be as expansive as the whole network. Each application may decide how "local" its interactions need to be. Consider a construction site supervisor that wants to monitor average concrete cure rates. He may limit the concrete sensors to those that lie within a circle of radius five meters around him or he may choose to receive the average concrete cure rates over all the pads throughout the site. The scene for the latter case would be specified as "all concrete cure rate sensors within the construction site boundaries." In this example, "within 5m" or "within the construction site boundaries" specifies the threshold. Finally, in any scene definition, as the mobile device moves among the sensors, a scene *specification* stays the same, but the data sources belonging to the scene may change.

A similar concept was explored in mobile ad hoc networks in the network abstractions model [24] which allows applications to provide metrics over paths of network connectivity. 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 very expressive, making it difficult for the novice application developer to specify simple metrics using the programming interface. In addition, it is too complex for operation on lightweight sensor nodes. More recently, similar concepts have been explored in the context of sensor networks. Hood [29] allows sensor nodes to define neighborhoods of coordination around themselves based on hop counts and other network properties. Similarly, Abstract Regions [28] allows applications to define regions of coordination but couples the abstraction with programming constructs that allow applications to issue operations over the regions. While both approaches have shown promise in coordinating traditional sensor network applications like object tracking and contour finding, the approaches do not directly consider dynamics and both require constant proactive behavior of any sensor that may belong to a region.

Mobicast [23] defines a message dissemination algorithm for pushing messages to nodes that fall in a region in front of a moving target. It provides “just-in-time delivery” to allow nodes to sleep until they need to receive a message by predicting the movement pattern of the target. MobiQuery [19] also supports spatiotemporal queries in a wireless sensor network and allows a query area to respond to a user’s announced motion profile. The scene concept, on the other hand, seeks to minimize the overall communication cost incurred by having a lightweight collection algorithm that reacts directly to a user’s actual movement.

3.1.1 A Programming Interface for Scene Definition

To present the dynamic scene construct to the application developer, we build a simple API that includes built-in general-purpose metrics (e.g., hop count, distance, etc.) and provides a straightforward mechanism for developers to insert additional metrics. Figure 3 shows the API for the `Scene` class.

```
class Scene{
   public Scene(Constraints[] c);
   public SceneView getSceneView();
}
```

Figure 3. The API for the `Scene` class

From the application’s perspective, the scene is a dynamic data structure containing a set of qualified sensors which are constrained as specified by a list of `constraints`, and accessed through a `SceneView`. Each constraint is a triple ⟨`Metric` *m,* `Aggregator` *a,* `Threshold` *t*⟩, and, to be included in a `Scene`, a node’s value for the metric must meet these constraints. The `getSceneView()` method provides access to the data sources in the `Scene` and separates the contents of a scene from the access to those contents. Thus, a user can specify a scene and seamlessly access the relevant data sources, even as he moves and the particular sources change.

Figure 4 shows some examples of how scenes may be specified. These examples include restricting the scene by the number of hops allowed, the required battery power on each participating node, or the maximum physical distance. We note that battery power *by itself* may not be a good metric (since it does not convey any notion of locality), but the use of multiple constraints in a scene definition allows it to be naturally and easily combined with other metrics (such as distance or hop count, to ensure that the query stops propagating).

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

Figure 4. Example Scene Definitions

As one example, SCENE_HOP_COUNT is a metric explicitly defined within the DAIS architecture, and it has the effect of assigning a value of one to each network link. Therefore, using the built-in SCENE_SUM aggregator, the application can build a hop count scene that sums the number of hops a message takes and only includes in the scene nodes that are within the maximum number of hops as specified by the threshold. For example, the following code defines a scene that includes every sensor within three hops of the declaring device:

```
Scene s = new Scene( { new Constraint(Scene.SCENE_HOP_COUNT,
                                      Scene.SCENE_SUM,
                                      new IntegerThreshold(3) } );
```

3.2 Programming Dynamic Behaviors

In the previous section, we defined how a programmer declares a scene to operate over. In this section, we describe how the application interacts with the scene through the `SceneView` interface.

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: `sendQuery()` (for one-time queries) and `registerQuery()` (for persistent queries). We also include versions of these two methods that request processing of the retrieved data before the result is handed back to the application. These methods are `sendAggQuery()` and `registerAggQuery()`. These methods employ the same `Aggregator` interface used in the scene specifications which reduces the burden on a DAIS programmer. In our current prototype, the aggregator specifies data aggregation across a scene (i.e., it provides *spatial aggregation*), though integration of *temporal aggregation* should be straightforward and is left for future work.

These operations and brief descriptions of their behavior are shown in Figure 5. To provide the functionality described in the API, the `SceneView` object keeps a list of live queries and a list of listeners as its private members. The information in the network is accessed on-demand through local interactions between mobile devices and the sensors dispersed throughout the scene. However, when no queries are active, no communication is required among the devices and/or sensors *at all*, in contrast to existing approaches which require constant proactive behavior.

Returning to the construction site applications, we give an example of each query type:

- `sendQuery`: The application asks for one location reading for each crane in the construction site.
- `registerQuery`: The application asks for crane location readings every 30s.
- `sendAggQuery`: The application asks for the average temperature from concrete sensors (as a measure of the cure rate) in the scene.
- `registerAggQuery`: The application asks for the average temperature from concrete sensors in the scene every five minutes. (The protocol implementing such a query may differ for different frequencies. For high-frequency queries, the middleware pushes the proactiveness to the sensors. For low-frequency queries, i.e., on the order of minutes, the middleware sends a one-time query each time, especially when the user is significantly mobile.)

The query model includes three additional abstractions: the `Query`, the `QueryResult`, and the `ResultListener`. The declarative specification of a `Query` allows the programmer to describe the data he

Operation	Description
`void sendQuery(Query q, ResultListener r)`	Sends a one-time query to the `Scene`. The result listener receives results from this query.
`int registerQuery(Query q, ResultListener r, int frequency)`	Registers a persistent query on the `Scene`. The frequency indicates how often readings come from each qualified sensor in the scene. The method returns a receipt that can be used to cancel the registration when desired.
`void sendAggQuery(Query q, ResultListener r, Aggregator a)`	Sends a one-time query, requesting in-network aggregation. The user provides the aggregation function.
`int registerAggQuery(Query q, ResultListener r, int frequency, Aggregator a)`	Registers a persistent query on the `Scene`, requesting in-network aggregation. The user provides the aggregation function. The method returns a receipt that can be used to cancel the registration when desired.
`void deregisterQuery(int receipt)`	Stops the registered query referenced by the receipt.

Figure 5. `SceneView` API operations

wants, without requiring him to specify how it should be obtained. Therefore, the query processing layer can change how it runs a query, without requiring the query itself to be changed. In our initial prototype, a `Query` provides a declaration of a simple data type provided by a sensor. To ease the programming task, the types enabled are built-in as constants within the middleware, so building a query is as easy as selecting a `Metric` for the scene. More complicated data types are constructed using virtual sensors. Within the middleware, each data type must also define how each style of aggregation is performed on the type (e.g., what it means to average the data). For common cases (e.g., numerical data), these definitions are built into the base class. While we omit the specifics for brevity, the interface for introducing new data types (as well as the interfaces for introducing new aggregators or new metrics) is quite simple and makes every effort to guide a novice programmer in creating correct constructs.
A `ResultListener` is registered to receive the results of each query. When the result is ready, the middleware calls the `resultReceived(QueryResult)` method for the result listeners to forward the results to the application. This achieves an asynchronous and nonblocking implementation, so that the application can perform other tasks in case no response comes back to the query or the response for the query is delayed.

4 Implementation Details

In this section, we describe the internals of the DAIS middleware that provides these programming abstractions described above. Figure 6 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 3. We first give brief details of the top layers of this design, relating them to our discussion of the API in Section 2. We then discuss the idea of the strategy pattern, its importance to the overall design of the system, and how our middleware makes use of this concept. Finally, we will detail one example realization of the scene abstraction on the sensors.

4.1 Application Programming Interface

The `Scene` object is created through the API provided to the application developer, as discussed in Section 3. That is, when the application needs to query the Scene, it calls the `getSceneView()` method which returns a handle to a `SceneView`. Once the application has this access to the `Scene`, it is free to

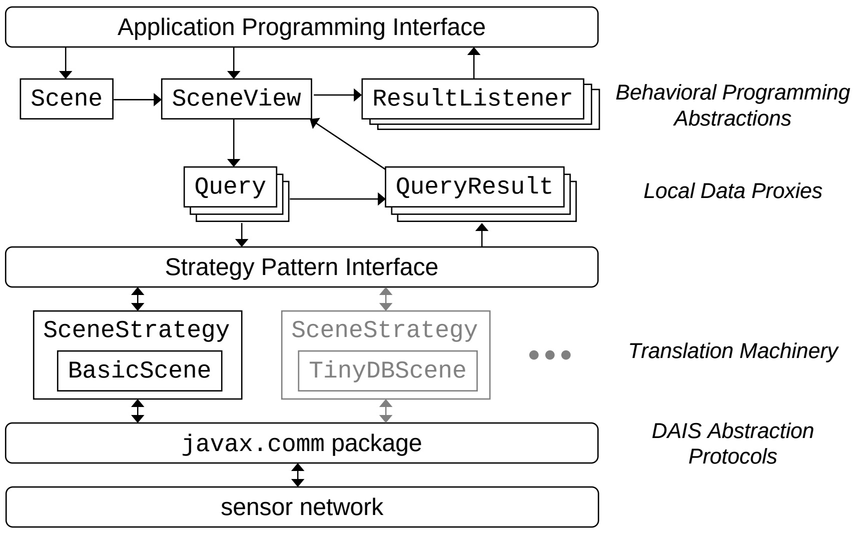


Figure 6. Simplified object diagram for DAIS

send a `Query` over the `Scene`. 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.

When the underlying protocol implementation returns a result to a query, the `QueryResult` is forwarded to the `SceneView`, which uses its local data structure to deliver the `QueryResult` to the appropriate `ResultListener`, by invoking the `resultReceived` method in the `ResultListener` interface. The local data proxies (i.e., the `Query` and `QueryResult`) mediate between the `Scene` and the `SceneView` APIs and the protocol implementations that underlie the middleware. As shown in Figure 6, 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*.

4.2 Strategy Pattern Interface

Our middleware makes use of the *strategy pattern* [6], 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.

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.

For now, we provide only a single implementation of the strategy, the `BasicScene`. This is a simple, greedy scheme in which all data aggregation is performed locally. We have chosen this as a first step to provide a quick prototype of the entire middleware. Other communication approaches can be swapped in for the `BasicScene` (for example one built around TinyDB [21] or directed diffusion [14], 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 plug-ins 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 our implementation of the portion of the `BasicScene` that must run on the sensor nodes that respond to a client application's queries. This serves as just one example of a particular implementation of the `SceneStrategy`.

4.3 Realizing the Scene Implementation on Resource Constrained Sensors

In DAIS, sensor components have been developed for the Crossbow Mica2 mote platform [2]. Our initial implementation is written for TinyOS [13] in the nesC language [7] and helps the `BasicScene` strategy conform to the `SceneStrategy` interface described previously.

4.3.1 Building Scenes

In nesC, an application consists of *modules* wired together via shared interfaces to form *configurations*. We have created several components that form the fundamental functionality of the *Scene configuration*. Figure 7 abstractly depicts the necessary components and the interfaces they share.

This implementation functions as a routing component on each node, receiving each incoming message from the radio 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 away from it and *uses* an interface if the arrow points towards it. If a component provides an interface, it must implement all of the *commands* in the interface, and if a component uses an interface, it can call any commands in the interface and must handle all *events* generated by the interface.

The `Scene` configuration uses the ReceiveMsg interface (provided in TinyOS) which allows the component to receive messages incoming (by handling the `receive` event). The structure of the messages received through this process is shown in Figure 8.

The message contains two constants that instruct the `SceneM` component in processing the scene calculation. The first indicates the property used to construct the scene (e.g., SCENE_DISTANCE from Figure 4). The second indicates the aggregation function to be used (e.g., SCENE_MIN from Figure 4). The use of these constants makes the implementation a bit more inflexible because the set of metrics that can be used in a network must be known on the sensors *a priori*, but the approach prevents messages from having to carry the code that defines the evaluation. The `metricValue` in the `SceneMsg` allows the Scene building process to propagate state. The `previousHop` in the `SceneMsg` allows this node to know its parent in the routing tree, which is important in maintaining the scene as the requester moves through

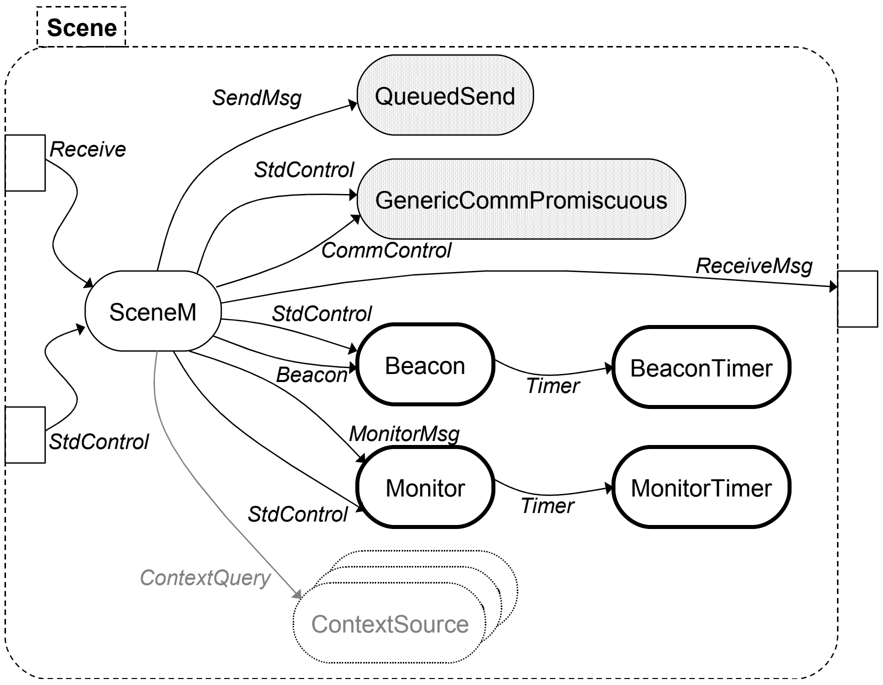


Figure 7. Implementation of the `Scene` functionality on sensors

the network. The `persistent` flag indicates if the query is long-lived. Finally, `data` carries the actual query. In this implementation of scene construction, our messages only carry a single scene constraint at a time due to limitations in the recommended size of a TinyOS message. Future work will investigate the use multiple messages to carry complete scene specifications that contain multiple constraints.

When the `SceneM` module receives a message, it first checks its table of recently received message IDs to determine if this is a duplicate. If this node has not processed the message before, it determines whether or not the local node should be considered as part of the scene. To do so, the `SceneM` implementation must calculate the metric value for this node based on the `metricValue` field in the received message and the metric and aggregator types specified in the message. Because the `metric` and `aggregator` fields in the packets are constants, the `SceneM` can lookup their meanings in a static table and determine how to calculate the new metric value. Depending on the metric type, this may require the use of *ContextSource*s. For example, a hop-count based scene requires no context sources; when looked up in the local table, SCENE_HOP_COUNT indicates the local metric value is "1" and the aggregator SCENE_SUM indicates that the value "1" should be added to the `metricValue` carried in the message. On the other hand, when looked up in the local table, SCENE_DISTANCE indicates the local metric value is the node's location, which in this case is implemented as a *ContextSource* that stores the node's (static) location (i.e., `LocationSource`). This value retrieval occurs through `SceneM` calling the `query` command in the ContextQuery interface. When the local value for the metric has been retrieved from the appropriate *ContextSource*, the `aggregator` from the message is looked up and the appropriate function called. In this case, the appropriate function calculates the distance between this node (retrieved from the `LocationSource`) and the originating node (carried in the `metricValue` field in the messages). The new value for the metric is compared against the value in the `threshold` field in the message. If the new metric value does not satisfy the threshold, then this node is not within the scene and the message is ignored.

```
typedef struct SceneMsg{
   uint16_t seqNo //message sequence number
   uint8_t metric //constant selector of metric
   uint8_t metricAggregator //constant selector of aggregator
   uint16_t metricValue //current calculated value of metric
   uint16_t threshold //cutoff for metric calculation
   uint16_t previousHop //the parent of this node in the tree
   uint8_t persistent //whether or not the query is persistent
   uint8_t data [(TOSH_DATA_LENGTH-11)] //the query
}
```

Figure 8. `SceneMsg` **Definition**

If this node is within the scene, the message is forwarded to allow inclusion of additional nodes. To do this, a new message is created with the same `seqNo`, `metric`, `aggregator`, `threshold`, `persistent` flag, and `data` portion. 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 this node adding one to the previous `metricValue`), while in the case of `SCENE_DISTANCE`, the `metricValue` is always the location of the originating node. When the message to be forwarded has been constructed, `SceneM` calls the `sendMsg` command on the `QueuedSend` component using the TOS_BCAST_ADDR as the destination.

After forwarding the message, the node must also pass the data query to the "application" through the Receive interface. In this case, the application is our `QueryProcessor`, described in more detail next.

If the query is a persistent query (i.e, the `persistent` flag in the message is `true`, then the `SceneM` module must monitor changes in the scene membership that might cause it to no longer be a member. To accomplish this, the scene must monitor incoming beacon messages for this scene from the parent (as indicated by the `previousHop` field in the message). Such messages are also received in `SceneM` through the ReceiveMsg interface. `SceneM` passes them to the `Monitor` module through the MonitorMsg interface. The `Monitor` module uses incoming beacon messages from the parent, information about the scene (from the initial message), and information from the context sources to monitor whether this 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` module generates an event that is handled by `SceneM` that ultimately ceases the node's participation in the scene. This includes signaling a `receive` event on the Receive interface connection to the `QueryProcessor`. This event passes a null message that indicates to the `QueryProcessor` to cease evaluating the noted persistent query. Finally, the `Beacon` module shown in Figure 7 is also activated when a persistent query is received. This module periodically transmits an update for this node's metric value for the scene. As the `Monitor` module detects changes in the metric value (based on the messages from its parent), the metric value is updated (through the `SceneM` module). When the `Monitor` module detects that the node is no longer part of the user's scene, the `SceneM` module shuts down the `Beacon` module.

4.3.2 Processing Queries

Once the `SceneM` module described above determines that the node is in the scene, it passes the received message to the `QueryProcessor` component shown in Figure 9. In our implementation, this component provides the functionality shown at the top layer of the sensor portion of the architecture in Figure 2. The query arrives in the `QueryProcessor` through the `receive` event of the Receive interface. If the query

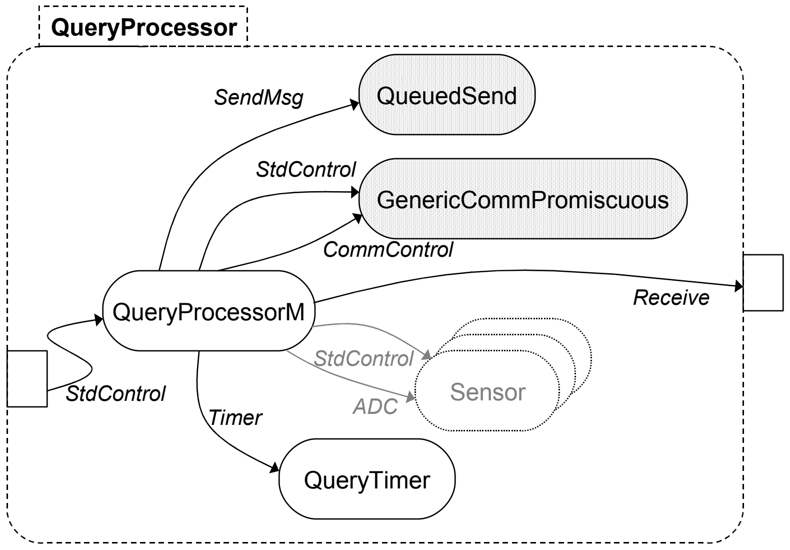


Figure 9. Implementation of the `QueryProcessor` functionality on sensors

is a one-time query (i.e., if the `persistent` flag is false), 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 `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, 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 `SceneM` module 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`.

5 A Case Study in Scene Definition

The metrics used to specify scenes can be divided into two categories. The first defines scenes based on properties of *network paths*, either defined by characteristics of links (e.g., latency) or devices (e.g., battery power). The second category defines scenes based on *physical characteristics* of the environment (e.g., location or temperature).

The ability to query a network based on such physical characteristics is important to many applications. However, the attempt to use a physical characteristic such as location to calculate network paths is plagued by the *C-shaped network problem.* Consider the network shown in Figure 10. In this case, nodes A and B are within 50 meters of each other, yet a discovery from A to B must leave the region of radius 50 meters surrounding A in order to find B. The only way to guarantee that every device within a distance radius is discovered is to flood the entire network. The network abstractions model [24] 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 [28] implicitly addresses this issue by enabling only *geographic filters* on neighborhoods defined by hop count. Hood [29] avoids this trouble altogether by limiting collection neighborhoods to one-hop regions and arguing that such regions meet the demands of current applications.

In our operational environment, however, a user may not be in direct communication range of the sensors from which he needs to gather information. On a construction site, safety applications may dictate that each user has information about a region larger than a sensor's communication radius, for example to monitor the presence and movement of hazardous materials [15]. For this reason, the scene abstraction focuses on building the best multi-hop neighborhoods possible. For metrics that measure physical characteristics (as opposed to network or device characteristics), the question remains as to how to handle the ambiguity separating the natural specification (e.g., "all devices within 200 meters") and the ability of a protocol to satisfy that specification. 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 Figure 10, our approach may not find some members of the specified scene even though they are transitively connected.

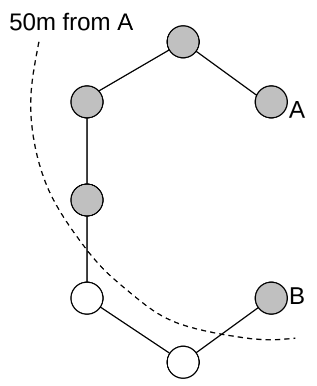
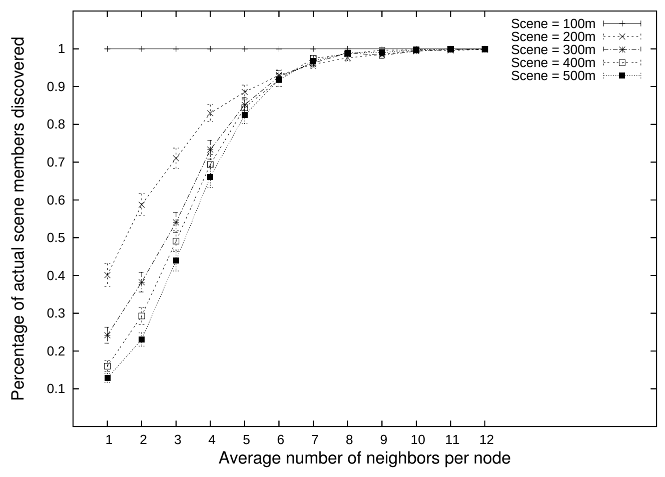


Figure 10. A C-shaped network

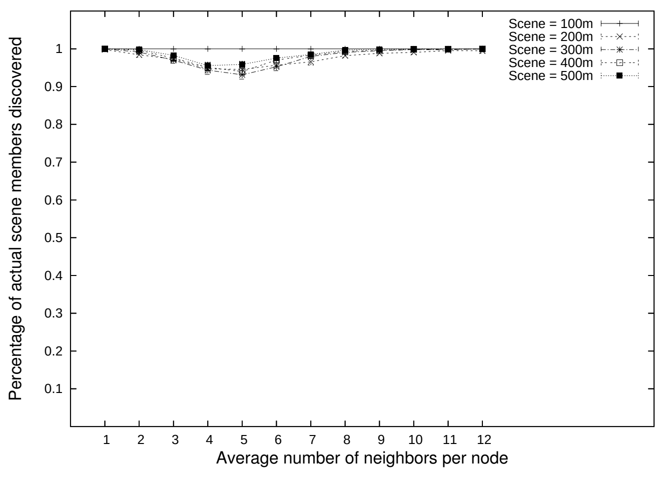
Figures 11(a) and (b) show the results of experiments that demonstrate the ramifications of this design decision. In these experiments, we generated random network topologies in a 1000m^2 space with the following parameters. The number of nodes was randomly selected to be between 20 and 400, and each node was placed randomly (and independently). We used a communication radius of 100m, i.e., any two nodes within 100m of each other were considered "neighbors." We justify the use of a static, circular communication range without fading or wireless channel considerations based on the combination of the fact that the necessary randomness was captured in the placement of the nodes and the fact that we used each topology only for a single instant of time. 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 approximately 500 samples, and 95% confidence intervals are given. For each sample, one node was randomly selected to request a scene of the specified size.

Figure 11(a) shows the percentage of *actual* members of the specified scene that were discovered by our scene construction protocol. In this case, *actual* scene members includes everyone within the specified physical distance radius, even nodes to which no network connectivity existed. As the graph shows, at low network density, the quality of the scene construction was poor, especially as the physical size of the scene increased. This is due to the fact that the network was so sparsely connected that it was unlikely that nodes were able to communicate, especially when they desired to find other nodes at large distances. However, with increasing density (i.e., when nodes had, on average, six or more neighbors), our protocol found more than 90% of the actual scene members.

Figure 11(b) demonstrates even stronger motivation for approximating scenes that are based on physical characteristics. This graph limits the error expressed to only those scene members that were not discovered but were connected to the requester by a finite number of network hops. The percentage of scene members that *were not* discovered by our method is never more than 10% and is usually close to 0. The valley in this graph (from densities with an average of three to six neighbors) corresponds



(a)



(b)

Figure 11. Accuracy of location-based `Scene` calculations

to cases when the network was largely connected, but the connections were sparse. In these situations, roundabout paths may exist between nodes when more direct routes do not. To the left of the valley, the network was largely disconnected, so our approach did not miss many connected scene members. To the right of the valley, the network was much more connected, and the direct approach is quite successful.

The results depicted in Figure 11 show that our approach tends to find the vast majority of the members of a scene under reasonable conditions. For this reason, DAIS favors the simplicity of natural scene specifications over complete accuracy of scene membership.

6 Related Work

Systems designed to address the specific challenges posed by sensor networks and/or pervasive 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 [31], and the DAIS platform attempts to follow these guidelines. Other projects have also undertaken similar efforts, and we highlight a few of these systems.

As one example of a middleware for pervasive computing, GAIA [25] introduces *Active Spaces* that can be programmed into pervasive computing applications. However, the model assumes a centralized and heavyweight system structure which is in direct opposition to the goal of middleware for coordinating with wireless sensor networks.

Projects targeted directly for sensor networks have often explored representing the sensor network as a database. Two demonstrative examples are TinyDB [21] and Cougar [30]. Generally these approaches enable applications with data requests that flow out from a central point (i.e., a base station) and create routing trees to funnel replies back to this root. Much of the work within these approaches focuses on performing intelligent in network aggregation and routing to reduce the overall energy cost while still keeping the semantic value of data high.

VM* [17] provides a virtual machine approach directed at handling device heterogeneity. While this is also an important concern in DAIS, VM* assumes situations where the application and its needs can be known in advance so that the VM deployed can be optimized with respect to the application and device. TinyGALS [1] 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. MiLAN [10] aims to enable applications to control the network's resource usage and allocation to optimally tune the performance of an entire sensor network through the definition of application policies that are enacted on the network. 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 [8] 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). The Sensor Network Application Construction Kit (SNACK) [9] consists of a set of libraries and a compiler that makes it possible to write very simple application descriptions that specify sophisticated behavior. Agilla [5] is an agent based middleware that allows applications to inject agents into the sensor network that migrate intelligently to carry out the applications' tasks.

DAIS differs from these approaches in that it perceives the sensor network not as a data repository but as an instrumented environment that has the capacity to augment a user's pervasive computing experience. Similarly, TinyLime [3] is a tuple space based middleware that enables mobile computing devices to interact with sensor data in a manner decoupled in both space and time. TinyLime provides only single-hop connections to sensors and assumes that the sensors do not communicate among themselves. This effectively places all of the burden of aggregation on the shoulders of the application developer.

7 Conclusions

In this paper, we have described DAIS, a tiered middleware that allows developers to create lightweight applications that run on client devices (e.g., laptops or PDAs) and allow users to interact directly with an immersive sensor environment. DAIS defines a set of programming constructs centered on the scene abstraction. A scene is a localized but dynamic view of the sensor network that adapts to changes as the user moves through his environment. By abstracting the process of selecting sensors to interact with, DAIS enables applications to make intelligent tradeoffs regarding the properties of the selected sensors and their communication links without having to directly deal with low-level programming concerns.

Using the scene abstraction, DAIS provides a wrapper for low-level data acquisition and aggregation, allowing applications to use a high-level, asynchronous query and response mechanism for retrieving data from the environment. As necessitated by the unpredictable sensor environment, these interactions are data-centric, thereby directly leveraging application knowledge within the communication process.

While the focus of this paper is programming abstractions and the ensuing simplification of development, we also demonstrated the accuracy with which scenes can be built around location information. Future work will include a similar evaluation for different metrics and a complete network performance evaluation that considers scene dynamics and measures query response times, energy usage, and overall communication overhead. Other future work will include the integration of additional strategies for scene construction and the ability to dynamically update the code on sensor nodes to include newly introduced metrics and aggregators.

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.

References

[1] 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.

[2] Crossbow Technologies, Inc. `http://www.xbow.com`, 2005.

[3] 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.

[4] D. Estrin, D. Culler, K. Pister, and G. Sukhatme. Connecting the physical world with pervasive networks. *IEEE Pervasive Computing*, 1(1):59–69, 2002.

[5] 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.

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

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] J. Hellerstein, W. Hong, S. Madden, and K. Stanek. Beyond average: Toward sophisticated sensing with queries. In *Proc. of the 2nd Int'l. Workshop on Information Processing in Sensor Networks*, pages 63–79, 2003.

[12] M. Hewish. Reformatting fighter tactics. *Jane's International Defense Review*, June 2001.

[13] 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.

[14] 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.

[15] C. Julien, J. Hammer, and W.J. O'Brien. A dynamic architecture for lightweight decision support in mobile sensor networks. In *Proc. of the Wkshp. on Building Software for Pervasive Comp.*, 2005.

[16] 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.

[17] 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.

[18] K. Lorincz, D. Malan, T. Fulford-Jones, A. Nawoj, A. Clavel, V. Shnayder, G. Mainland, M. Welsh, and S. Moulton. Sensor networks for emergency response: Challenges and opportunities. *IEEE Pervasive Computing*, 3(4), 2004.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] Huang Q, 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] 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.

[25] 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.

[26] 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.

[27] M. Weiser. The computer for the twenty-first century. *Scientific American*, 265(3):94–101, 1991.

[28] 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.

[29] 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.

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

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