Virtual sensors: Heterogeneous aggregation in pervasive networks

Sanem Kabadayi
Christine Julien
Jason Trujillo

TR-UTEDGE-2006-009

Virtual sensors: Heterogeneous aggregation in pervasive networks

Sanem Kabadayı *, Christine Julien, Jason Trujillo

Department of Electrical and Computer Engineering, 1 University Station C5000, The University of Texas at Austin, Austin, Texas 78712-0240, USA

Abstract

Sensor networks are increasingly used to provide pervasive environments. These deployment environments entail embedded sensing devices that have limited processing, memory, and battery power. It is therefore essential to devise efficient approaches that can collect low-level sensed data and transform it to a higher, more abstract measurement to relay to the user. This paper introduces the virtual sensors abstraction that enables an application developer to create software sensors that can apply user-specified functions on different types of data to provide a higher-level measurement that is not an intrinsic measurement provided by the physical sensors in the network. We evaluate our implementation of the virtual sensors model in two diverse example domains.

Key words: Sensor networks, middleware, context-awareness, coordination, declarative languages

1 Introduction

Sensor networks are an integral component of pervasive computing environments. Currently, existing deployments of sensor networks are application-specific, where the nodes are statically deployed for a particular task. We target pervasive environments 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 [1], intelligent construction sites [2], battlefield scenarios [3], and first-responder deployments [4].

* Corresponding author. Tel.: +1 512 232-5671; fax: +1 512 471-5120.
E-mail addresses: {s.kabadayi, c.julien, jasontru}@mail.utexas.edu

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 [5], in which sensor networks must play an integral role [6].

The data collection schemes used in existing deployments of sensor networks commonly require sensors to relay raw data to sink nodes to perform further processing. For the sensing devices, communication is much more expensive than local computation, thus the approaches used in existing deployments lead to short network lifetimes. Furthermore, the throughput at each node decreases as the network scales due to redundant broadcasts, leading to inefficient use of the network bandwidth. Sensor network aggregation mechanisms [7–10] offer in-network processing algorithms that are successful in limiting resource usage by aggregating data in a tree (i.e., calculating the updated value at each node along the tree, on the way back to the base station). However, these approaches support only standard mathematical operators (e.g., minimum, count, and average) over homogeneous types. To support pervasive environments, sensor networks will need to support localized cooperation of sensor nodes to perform complicated 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.

Sensor networks for these environments will also need to provide reusability. Current efforts offer solutions that are specific to a particular application (e.g., once the sensor network has been tasked to do habitat monitoring, that is all it is expected to do), but the future will see multipurpose networks deployed to support numerous applications whose natures may not be known at the time the network is deployed. The cost of physically visiting each sensor to reprogram it is prohibitive, and therefore the ability to remotely and dynamically tailor sensor networks to particular applications will be essential. This paper tackles exactly these challenges through the introduction of *virtual sensors*.

In this work, 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. More specifically, an in-network aggregation mechanism for pervasive environments must:

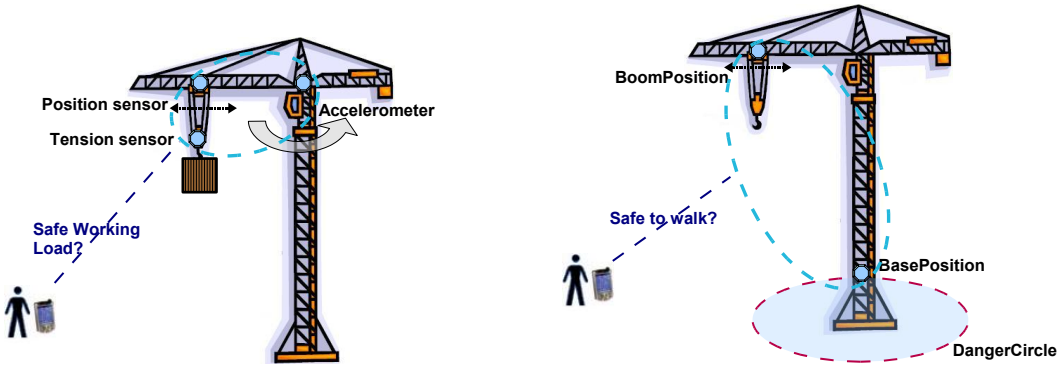
- *Have a simple programming interface:* The interface should support data-centric requests from the user and enable fast deployment of applications by novice programmers.
- *Support multiple access points and coordination through multihop connec-*

tions: A truly decentralized approach needs to allow access from anywhere in the network and not rely on a designated gateway at which to perform aggregation. The large scale of pervasive computing networks further requires the ability to communicate through multihop connections.
- *Support heterogeneous devices:* Devices that will be used in pervasive applications may include RFIDs, smart cell phones, sensors from different vendors, etc.
- *Cope with unpredictable availability of resources:* Mobility and environmental factors affecting the wireless medium may cause certain resources to become unavailable at unexpected times.
- *Offer locality of interactions and access to local data:* In pervasive computing environments, the user often needs information from a specific region defined by an abstract notion of locality, and the interactions need to be kept local without having to go through a remote server or gateway.

All of these challenges call for an abstraction with a declarative interface that allows the user to specify what he needs from the network without having to specify how it should be obtained. This abstraction needs to adapt to applications' on-demand dynamic interactions while also accounting for resource constraints.

To address the issues outlined in our problem definition, we have created the virtual sensors model. A virtual sensor is a software sensor as opposed to a physical or hardware sensor, which builds on an initial version presented in [11]. 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. [12]. Signals from these individual physical sensors can be used in calculations within a virtual sensor to determine if the crane has exceeded its safe working load. Using fewer data types for ease of presentation, Fig. 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, Fig. 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.

The power of virtual sensors lies in the fact that the physical sensors used by the virtual sensor may be heterogeneous (in the case of our first exam-



(a) safe working load (b) danger circle

Fig. 1. Two different examples of virtual sensors for the construction site domain

ple, position, tension, and acceleration), and the virtual sensor can combine these different types of data to compute an abstract measurement (in our first example, safe working load). Another benefit of the virtual sensor is that it can be used to mask the *explicit* data sources (sensors) that provide data. A simple example would be a virtual position sensor that uses GPS when the necessary hardware is available on the local device but can switch to providing a position estimate based on the relative positions of other nearby (physical) location sensors if GPS is unavailable (e.g., inside a building).

The specific novel contributions of this work are at both the model and implementation level. We define a new virtual sensor model designed to abstract data from heterogeneous physical sensors by applying user-defined functions. We realize 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 is demonstrated to support applications in two different domains.

Section 2 of this paper examines related work. Section 3 describes our virtual sensor model. In Section 4, we provide a detailed description of the middleware implementation of this model. In Section 5, we discuss two examples of specifying a virtual sensor, drawn from diverse application domains, and Section 6 concludes.

2 Related Work

As the main goal of this work is to provide in-network aggregation in sensor networks in support of pervasive computing applications, we will overview a combination of related work in sensor networks and pervasive computing that have addressed components of our problem definition. Throughout the discussion, we highlight components of our stated problem definition that are

not completely satisfied by these existing systems.

The tiny embedded sensing devices that support pervasive computing environments have many constraints as discussed in Section 1. Limited energy is one of the major concerns and the starting point for addressing this problem is reducing communication in the network. The amount of communication each sensor node will have to perform can be reduced through the use of in-network aggregation, so that not all the results have to be relayed back to the sink requesting the data, but the necessary result is calculated on the way to the sink through the cooperation of local nodes. 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. That is, query processors running at each node process streams of sensor measurements in the same manner as the processing of streams in a database. Two demonstrative examples are TinyDB [8] and Cougar [10]. 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, which contradicts our goal of supporting multiple access points. In both approaches, data aggregation is specified using an SQL-like language. However, queries cannot be used to merge different data types, i.e., only homogeneous data aggregation is possible.

Compared to these approaches, directed diffusion [13] is a less centralized, data-centric system. Attribute-based naming and filtering provide access to sensor network data from multiple points inside or outside the network. Filtering is also used for in-network data aggregation. A disadvantage of directed diffusion is that it requires gradients to be set up from all the sources (that will be used in an aggregation) to the sink that requests this information. Gradient setup is expensive, and the best paths may be overused, depleting the battery power on some sensors. Also, alternate path maintenance is necessary, and interests need to be retransmitted if responses from sources are lost. This makes it difficult for directed diffusion to cope with unpredictable resource availability. In addition to being expensive, directed diffusion's predefined filters are not expressive enough to meet the demands stated in our problem definition.

Similar to directed diffusion in terms of being data-centric and enabling mobile computing devices to interact with sensor data in a manner decoupled in both space and time, TinyLime [14] gives the programmer the option of accessing a specific sensor's data on-demand, as well as the option of computing an aggregation over the continuously sampled data. However, TinyLime provides only single-hop connections to sensors (with the assumption that the sensors do not communicate among themselves) and only a data filtering function at the base station, directly contradicting the goal of supporting multihop connections that define a tailored and localized region for the query.

These middleware solutions for sensor networks typically provide data aggregation in the form of minimum, maximum, sum, and average. To support the abstraction of more complicated information from explicitly gathered information for context-aware systems, ontology-based solutions such as SensorML have been introduced. The Sensor Model Language (SensorML) [15] provides an XML schema for defining the geometric, dynamic, and observational characteristics of a sensor. The language introduces a "sensor group" composed of multiple sensors that together provide a collective observation. However, since SensorML is targeted for satellite-based high-power sensor systems used in remote imaging, it does not take power constraints into consideration, and the sensor definitions are too detailed for operation on lightweight sensors. Also, since data processing is not defined rigorously, SensorML cannot support the application of arbitrary and complex user-specified functions to different types of data available in an instrumented environment. SensorML places the XML descriptions on the external interface, not on the sensor nodes and does not support in-network processing of sensor descriptions. While XML is human-readable, easy-to-parse, and system-independent, it is verbose, and using XML to access stored data can lead to long access and processing times.

Instead of defining device characteristics, CODE [16], a description language for Wireless Collaborating Objects (WCO), focuses on describing *services* in a sensor network. CODE stores the XML descriptions at the gateway level, with the sensor nodes storing and processing a binary version. As an example, a user requesting a localization service sends out a broadcast message. Statically-placed beacon nodes that hear this broadcast organize into a group and deliver the service to the user. In CODE, a sensor is described in terms of its hardware, software, mobility, location, groups it belongs to, context information (such as neighbors), services it offers, type of measurements it performs, and quality of service (range, resolution, accuracy). Groups are described by the number of members, the initiator of the group, the group members, the leader, and the offered services. This approach makes the grouping difficult to maintain and requires explicit knowledge of the group members. It also requires a central system that conveys information to the beacons about group assignments and environmental conditions, and the beacons make use of a fixed infrastructure. This makes CODE inflexible in terms of coping with unpredictable availability of resources.

More recently, *virtual nodes* have been introduced [17]. The set of physical nodes used to define the virtual node is specified using logical neighborhoods [18]. 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.

To summarize, the above systems either offer database approaches, do not support multiple points of entry, or do not support specification of compli-

cated functions (any nontrivial functions that the user specifies, that can be composed of numerous operations and make use of different data types). As such, existing systems do not provide the necessary constructs to support all of our needs, especially since we require support for heterogeneous pervasive computing applications. Hence, we have developed the virtual sensors model. The most important potential of the virtual sensors model is its ability to create arbitrary user-specified functions over heterogeneous data types.

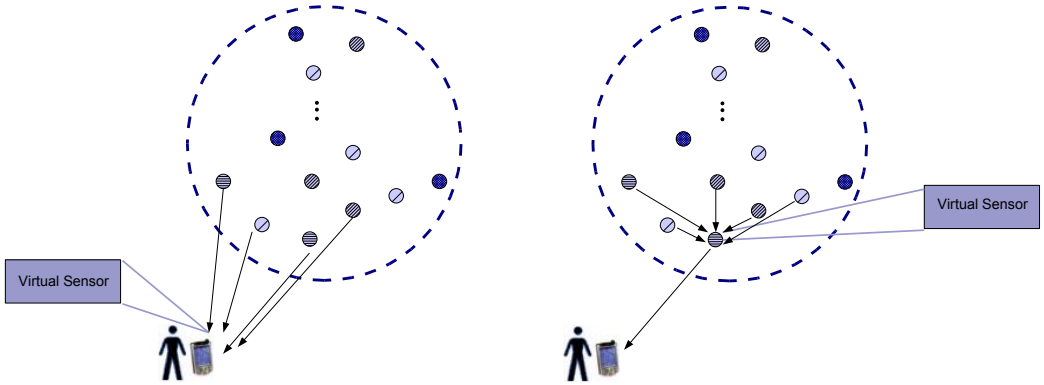
3 Virtual sensors model

In this section, we introduce the virtual sensors model, which we have developed to meet the challenges outlined in Section 1. We will first give an overview of the model. We will then give details of virtual sensor creation. To create a virtual sensor, it is essential to model the input physical data types, as well as the abstract type that will result from the aggregation. With the data types defined, the sensors that can provide these types need to be discovered. Since we do not want to flood the whole network with messages for sensor discovery when we are actually only interested in the data from nearby sensors, we make use of a scoping abstraction, the *scene*, to limit the reach of discovery messages. We end by discussing how virtual sensors are used and, if necessary, maintained in the face of environmental dynamics.

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 an abstraction called a *virtual sensor*. This abstraction offers generality and flexibility, since it has a declarative specification that offers the ability to specify any desired function and leaves the discovery of the physical data sources to the underlying communication layer. It also provides a higher level of abstraction to 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 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 to that of a physical sensor (from the application's perspective).



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

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

Fig. 2. Abstract depiction of a virtual sensor that uses n physical nodes

Fig. 2 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. 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.

3.2 Creating and deploying a virtual sensor

To create the virtual sensor, our model requires a declarative specification. This declarative specification allows a programmer to describe the behavior he wants to create, without requiring him to specify the underlying details of how it should be constructed. Most importantly, 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:

- *Input data types*: Physical (low-level) data types required to compute the desired abstract measurement. Each input data type also includes the number of different sensors (which could be *one*, *two*, *all*, etc.) the virtual sensor

would like to obtain this data type from. 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 specific (possibly heterogeneous) input data to calculate the desired measurement.
- *Resulting data type*: The abstract measurement type that is a result of the aggregation.
- *Aggregation frequency*: The frequency with which this aggregation should be made. This frequency 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 aggregation frequency, the virtual sensor "samples" each physical sensor that it uses and aggregates these results.

By providing these virtual sensor specifications, an application delegates physical sensor discovery to the virtual sensor (and to the framework that supports the virtual sensor). Therefore, if the data sources supporting the virtual sensor change over time, the virtual sensor adapts, but the application does not notice. This concept of data sources changing over time will be discussed in more detail in the following subsections.

3.2.1 Modeling data types that define a virtual sensor

In our model, the input data types carry simply the nature of the physical measure (e.g., "temperature") provided by a sensor. There could be a data type that is sometimes provided by a physical sensor and is sometimes provided by a virtual sensor (e.g., location provided by GPS (physical sensor) or by triangulation (virtual sensor)). From the application's perspective, this is only a single data type, and our model uses a data ontology (that may be application domain-specific) to describe these data types. The ontology is basically a simple listing of types. The application programmer can also insert new data types into the ontology to update or augment it over time. The implementation of the ontology is discussed in more detail in Section 4.

3.2.2 Scenes: locality for virtual sensors

As stated in Section 1, our goal is to support locality of interactions and access to local data. Thus, it is necessary to define the region from which it is allowable to select physical sensors to support a virtual sensor. In this section, we describe the *scene* abstraction [19], which allows the virtual sensor specification to restrict the region from which its physical sensors can be chosen. This allows us to limit the reach of a virtual sensor to some small portion

of the network in the immediate vicinity of the client device. When a virtual sensor construction is requested, the necessary scene is created first, then the virtual sensor is constructed.

The programmer must specify the following three parameters for the scene: 1) the *metric*, a property of the network or environment that defines the cost of a connection (i.e., a property of hosts, links, or data); 2) the *path cost function*, a function (such as sum, average, minimum, maximum) that operates on a network path to calculate the cost of the path; and 3) the *threshold*, the value a path's cost must satisfy for that sensor to be a member of the scene. Scene construction 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]

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 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.

3.2.3 Formalization of the virtual sensors model

After specifying the constraints that build the scene (which consist of a metric definition and a maximum permissible cost for that metric), a client application would like to construct the virtual sensor using physical sensors from within the scene. That is:

> *Given the set of hosts S_α in the scene, the required physical data types D_1,*

[1] In the three-part notation: $\langle$`op` *quantified_variables* : *range* :: *expression*$\rangle$, 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*.

D_2, ..., D_n, the aggregation function F, and the resulting data type, D_{res}, the virtual function can be formalized as:

$D_{res} = F(D_1, D_2, ..., D_n)$, *where:*

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

In the above definition, the set S is the subset of S_α that defines which physical devices contribute to the virtual sensor. 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 Section 3.2 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. 1(b) requires two data types and one sensor of each type: a single crane base position and a single crane boom position. On the other hand, 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 sensor above) or *all*, indicating that all matching sensors in the scene should be polled. In the latter case, the count value for the data type is set to be exactly the cardinality of the scene, $|S_\alpha|$.

To summarize, the virtual sensor aggregation function (F) operates on the input data that is of the specified type to yield the specified output data type. This function evaluation takes place at every interval specified by the aggregation frequency.

3.2.4 Discovering sensors

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. At this interface, an ontology exists that defines the data types available to the application. Complex virtual sensors can be created by hand by a domain expert and their types added to the ontology. Most importantly, the application 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

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

for supporting physical sensors in the scene.

A key question that arises in associating sensors with each other to create the virtual sensor can be demonstrated using the construction site domain as an example: *"How do we know that the crane arm we selected is attached to the crane base we selected?"*

In this work, we make the assumption that applying scoping through the scene abstraction before applying the virtual sensor limits the scene to contain at most one virtual sensor of the type we want to construct. For the construction site domain, this would correspond to having only one crane in the scene. Future work will explore defining a virtual sensor's scope from the perspective of the physical sensors that comprise it instead of from the client device that creates it, further ensuring the appropriate relationship among the virtual sensor's defining physical sensors.

3.3 Using a virtual sensor

A virtual sensor can be defined at any time, but it is not used until the application queries it. An aggregation frequency has been defined in the virtual sensor's creation, but that does not mean that the virtual sensor sends updates according to that frequency once it has been defined. Analogous to physical sensors which are capable of taking readings but usually do not do so until some application instructs them to, virtual sensors do not send responses back until they are actually queried by the application. When no queries are active, no proactive behavior is required from the virtual sensor, other than periodic evaluation of its aggregation function over the input data.

3.3.1 Virtual sensor maintenance

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 pervasive environments are embedded (hence stationary), the application interacting with them runs on a device carried by a mobile user. Therefore, the device's connections to partic-

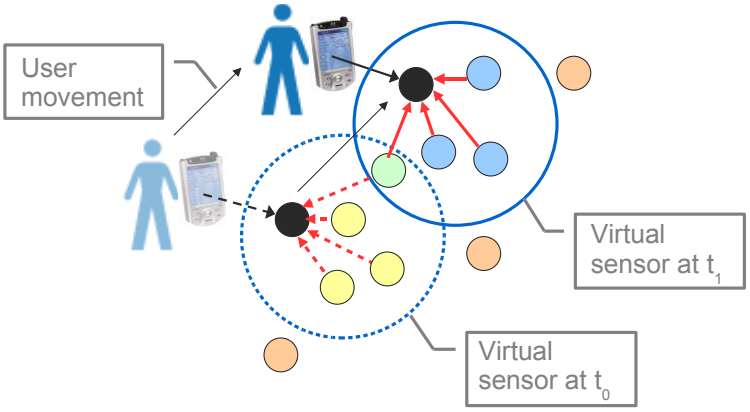


Fig. 3. The dynamics associated with user movement

ular sensors and the area from which the application desires to draw information (i.e., the scene) are subject to constant change. These changes need to be seamlessly handled without revealing the underlying dynamics to the user. The dynamics associated with user movement may cause the physical sensors that comprise the virtual sensor to change and are handled by the scene abstraction (see Fig. 3). Since the physical data sources that can contribute to a virtual sensor are restricted by the scene, any data source that satisfies the data requirements from that scene is a valid data source for the virtual sensor.

Different types of virtual sensors are maintained in different ways. For example, a virtual sensor that wants to average all of the concrete heat sensors (or trench strain sensors) in the scene does not really need to rediscover a new resource if one of the concrete heat sensors contributing to it dies (it does have to perform maintenance if the client moves); an average can still be calculated based on the remaining inputs. However, a virtual sensor that uses any crane arm location, crane base location, and crane boom location in the scene needs to rediscover a new resource if one of the physical sensors dies, since a result cannot be computed without the missing input data.

Going back to the location virtual sensor example we discussed before, it may be necessary to switch from the GPS on the local device (which is a physical sensor) to providing a position estimate based on the relative positions of other nearby physical location sensors if GPS service becomes unavailable (e.g., due to the unavailability of signals from satellites when the user enters a building). This switch from a physical sensor to a virtual sensor is not apparent to the application, and the middleware handles the mechanics using information expressed in the ontology.

To summarize, this dynamic maintenance allows the user to interact directly with a changing set of local information sources. The virtual sensor hides the underlying complexity from the user.

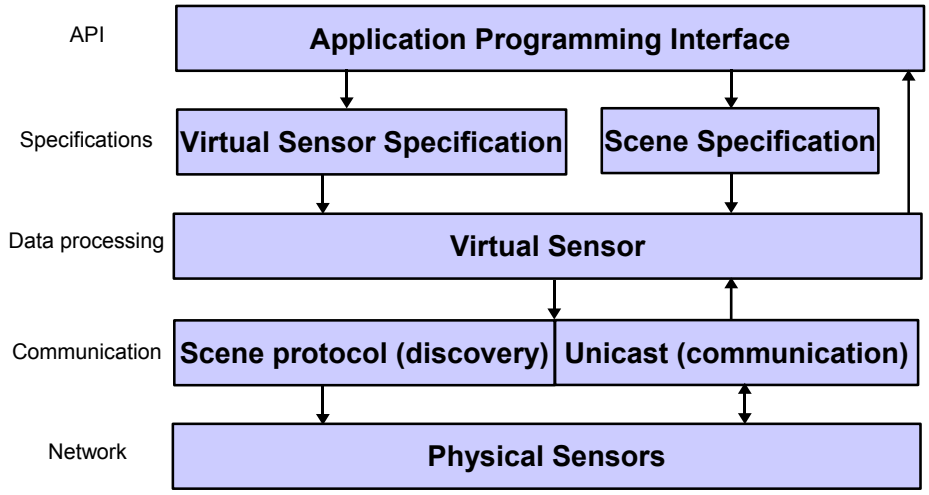


Fig. 4. Simplified object diagram for the virtual sensor middleware

4 A middleware implementation

In this section, we describe the implementation of the virtual sensors abstraction described above. We first give brief details of the top layers of this design, relating them to our discussion of creating and using virtual sensors in Section 3. Then we discuss the communication infrastructure, sensor discovery, and virtual sensor deployment. Fig. 4 depicts the middleware's simplified object diagram.

We will first give details of the application programming interface (API). Through this API, the virtual sensor and scene specifications are created. The virtual sensor that is constructed as a result of these specifications uses the scene protocol for physical sensor discovery, but it subsequently communicates with the physical sensors that make up the virtual sensor using simple unicast. The results from these physical sensors are aggregated in the virtual sensor.

4.1 Application programming interface

When the application needs to query the sensor network for a data type that is not provided intrinsically 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.

The `VirtualSensor` object keeps a list of live queries and a list of listeners as its private members. This allows a single virtual sensor to support 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.

To present the dynamic virtual sensor construct to the application developer, we build a simple API that includes built-in general-purpose data types (e.g., temperature, location, angle, etc.) and provides a straightforward mechanism for developers to insert additional data types.

Using the built-in data types, an application can query the data sources that provide these types. The application developer can define new data types that inherit the `DataType` class. Additionally, the application developer can define new virtual sensors by providing the aggregator function that must operate on these data types. This is explored in further detail in Section 5.

4.2 Sensor discovery

The application delegates discovery of physical sensors to a middleware that locates physical sensors based on the specified input data types. To be used by a `VirtualSensor`, a node must be able to supply at least one of the input data types specified in the creation of the virtual sensor.

In addition to using the scene abstraction to restrict the scope, the implementation uses the scene protocol (the built-in communication mechanism for the scene abstraction) to achieve sensor discovery and communication. Abstractly, this communication protocol initially broadcasts a data type requirement across exactly the sensors in the scene, and sensors that can provide that data type respond. Heuristics such as least latency, shortest path, etc., can be used to select a particular data source from many. This selection can be refreshed if the data source selected becomes unavailable. Our implementation's assumption is that any physical sensor in the scene that matches the virtual sensor's discovery request is as good as any other. Once a source is selected, unicast alone is used to communicate with it. The middleware uses the consistency frequency, `aggfreq` (i.e., the frequency with which physical sensors contributing to the virtual sensor are sampled and aggregated), to determine how often the sensor selections need to be to refreshed.

As described in Section 3, a virtual sensor specification requires the following four parameters: input data types, aggregator, resulting data type, and aggregation frequency. Currently, our prototype implementation relays only the input data types and the aggregation frequency to the sensors in a message for sensor discovery and for determining how often the sensors need to respond back. The other two parameters are used only on the Java side of the virtual sensor implementation that runs on the client device to perform the aggregation calculation. We have not yet implemented deploying the virtual sensor

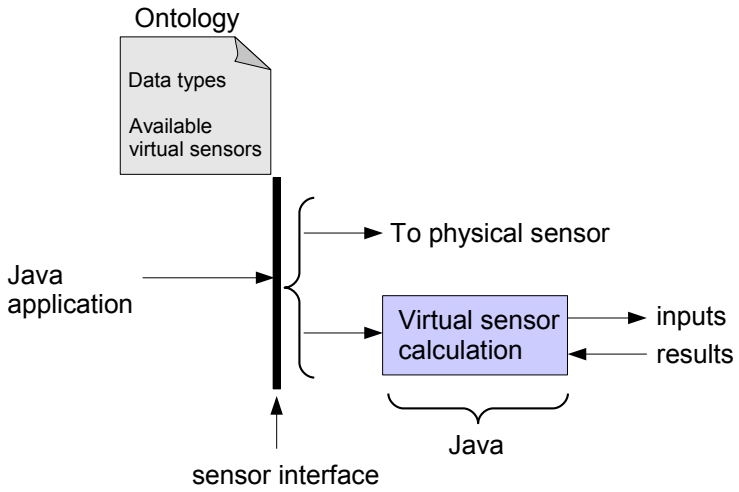


Fig. 5. Virtual sensor architecture

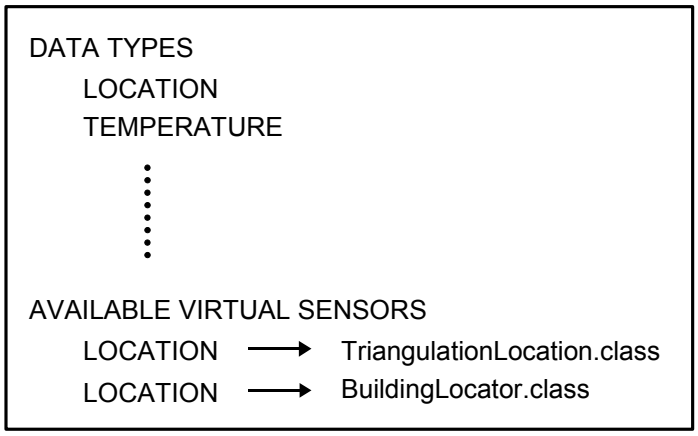


Fig. 6. Ontology

definition remotely. We expect to remove this limitation by adding a limited form of translation from Java to nesC code.

Fig. 5 shows the architecture for the virtual sensor implementation. A Java application sees a single interface that allows seamless access to both physical and virtual sensors. Which virtual sensors are available depends on which physical sensors are available. The ontology is currently a static list, but future work will explore making it dynamic to reflect the changes in the availability of certain data types in the network. As shown in Fig. 6, the ontology defines data types and the available virtual sensors. It is important to note that the developer or the user only sees what *functionality* is available, not what is deployed in the network (as physical or virtual sensors). The two different location virtual sensors in the ontology define different ways of obtaining the location.

Different types of virtual sensors are discovered in different ways. If a virtual sensor requires only one sensor (as specified by the number of sensors in input

data types) of a certain data type, all the sensors that provide that data type within the scene respond, however, these responses are filtered by the virtual sensor (e.g., the virtual sensor takes only the first response). Whereas, if all the sensors that can provide that data type are requested, the virtual sensor keeps and uses all the responses it received from the scene.

4.3 Interacting with virtual sensors

Since a virtual sensor offers the application an interface that is similar to that of a physical sensor, using a virtual sensor is analogous to extracting data from a physical sensor. The key idea is that we are not defining a new query model for virtual sensors; an application interacts with them in the same way that it interacts with physical sensors. The interesting part of the abstraction is what the middleware implementation for the virtual sensor has to do in order to make it appear to the application or user that this is in fact the case.

To demonstrate the functionality of the virtual sensor, we defined two simple query methods. The types of queries enabled on a virtual sensor can be classified into *one-time queries* (which return a single result from the virtual sensor) and *persistent queries* (which return periodic results from the virtual sensor). As with physical sensors, more complicated query languages can be implemented that incorporate the virtual sensors.

4.4 Deploying virtual sensors

As demonstrated by examples in the next section, the middleware currently translates portions of the virtual sensor specified by an application into low-level code (written in nesC [20]) (that provides the virtual sensor's functionality and communication and can run on TinyOS [21]) and evaluates the remainder in Java on the client device. Future work will provide the complete translation to make the implementation fully match the model and enable us to intelligently deploy virtual sensors to the sensor network itself.

4.4.1 Design for local and remote deployments of virtual sensors

In this section, we explore our design for local and remote deployments of virtual sensors. The virtual sensor code can either run locally 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. While this approach will require a small amount of our (general-purpose) middleware to run on every sensor in the network, we believe this is

a small price for dynamic reprogrammability.

The decision about where to deploy a virtual sensor should be based on the expense of communicating with the physical sensors that it comprises. 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. Future work will create heuristics within the middleware for automatically determining when the virtual sensor should be deployed remotely and where it should optimally be placed. We conjecture that remotely deploying virtual sensors when appropriate will be more efficient (require less communication) than creating all of the communication flows back to the user's device. This savings is precious in sensor networks as the nodes that have to route the messages contain limited battery power.

If the virtual sensor happens to be running remotely, a remote handle to the listener needs to be set up. In such cases, the middleware creates a proxy for the virtual sensor on the user's device. This proxy object runs within our middleware and uses a unicast routing protocol to connect to the remote virtual sensor and collect the information desired by the application. When the query's result is ready, this proxy makes a callback to the user's result listener either once (for a one-time query) or periodically (for a persistent query). It is important to note that the interactions do not change at all for remote deployments from the application's perspective; the only change is in the way the middleware handles them.

5 Example virtual sensors

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 multi-purpose networks of virtual sensors, we have chosen application examples from the following two different domains: a construction site and a first-responder deployment. As was shown in Fig. 4, 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 can be deployed to a sensor in the network.

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. 1(b) in Section 1). 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 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);
   }
}
```

Currently, the middleware 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.

Since nesC does not support non-primitive functions, when we implement remote deployments of virtual sensors, we may have to restrict the complexity of the operations that we perform on the sensor to, for example, not include the square root. Then, the proxy running on the client device (written in Java) would include a filter that knows to take the square root of the value returned from the network before returning it to the application. Alternatively, additional operations could be deployed with the virtual sensor (as done in Maté [22]).

5.2 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,

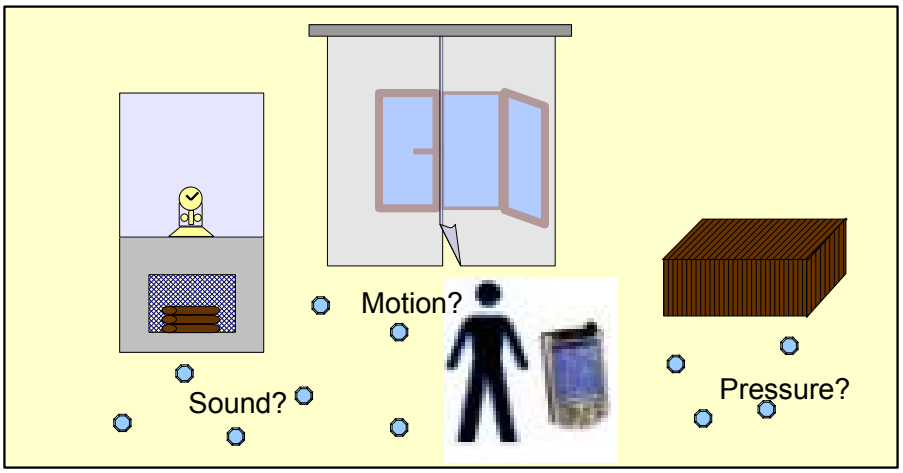


Fig. 7. Aware home example

pressure, and motion readings obtained from the room, as shown in Fig. 7. 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 [23]).

6 Future work and conclusions

In this paper, we have introduced the virtual sensors model implementation with respect to local deployment.

Future work will investigate the addition of proximity functions to the toolkit of virtual sensors. These proximity functions will define the relative relations of sensors to one another (in contrast to the relative relations of sensors to the user that the scene abstraction currently provides). An example would be the specification that all sensors be located on the same crane. 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.

As part of our short-term goals, we will implement remote deployment of virtual sensors. We will explore using Maté [22] or a similar system as a way of encapsulating mobile code in TinyOS messages to send additional complicated aggregation functions to remote sensors. We will further enhance this by creating intelligent algorithms for optimizing the deployment of virtual sensors based on available physical sensors and communication link qualities and making smart mobile virtual sensors that can follow the user or an event through the environment.

Other future work will 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 will 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).

We have defined a new virtual sensor model designed to abstract data from heterogeneous physical sensors by applying user-defined functions. To support virtual sensors, scenes provide a scoping or grouping abstraction. 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 have 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.

Acknowledgements

The authors thank the anonymous reviewers for their comments. We would also like to thank the Center for Excellence in Distributed Global Environments for providing research facilities and the collaborative environment and Adam Pridgen for his work on an earlier version of this model. This research was funded, in part, by the National Science Foundation (NSF), Grant # CNS-0620245. The conclusions herein are those of the authors and do not necessarily reflect the views of the sponsoring agencies.

References

[1] 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*, 1999, pp. 191–198.

[2] C. Julien, J. Hammer, and W. 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.

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

[4] 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*, vol. 3, no. 4, pp. 16–23, October–December 2004.

[5] M. Weiser, "The computer for the 21st century," *Scientific American*, vol. 265, no. 3, pp. 94–101, September 1991.

[6] D. Estrin, D. Culler, K. Pister, and G. Sukhatme, "Connecting the physical world with pervasive networks," *IEEE Pervasive Computing*, vol. 1, no. 1, pp. 59–69, January–March 2002.

[7] 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*, 2003, pp. 63–79.

[8] S. Madden, M. Franklin, J. Hellerstein, and W. Hong, "TinyDB: An acquisitional query processing system for sensor networks," *ACM Trans. on Database Systems*, vol. 30, no. 1, pp. 122–173, March 2005.

[9] N. Shrivastava, C. Buragohain, 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*, 2004, pp. 239–249.

[10] Y. Yao and J. Gehrke, "The cougar approach to in-network query processing in sensor networks," *ACM SIGMOD Record*, vol. 31, no. 3, pp. 9–18, September 2002.

[11] 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*, 2006, pp. 587–592.

[12] R. L. Neitzel, N. S. Seixas, and K. K. Ren, "A review of crane safety in the construction industry," *Applied Occupational and Environmental Hygiene*, vol. 16, no. 12, pp. 1106–1117, December 2001.

[13] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heideman, and F. Silva, "Directed diffusion for wireless sensor networking," *IEEE/ACM Transactions on Networking*, vol. 11, no. 1, pp. 2–16, February 2003.

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

[15] "Sensor Model Language (SensorML)," http://vast.uah.edu/SensorML, 2005.

[16] R. S. Marin-Perianu, J. Scholten, and P. J. M. Havinga, "CODE: Description language for wireless collaborating objects," in *Proc. of the 2nd Intelligent Sensors, Sensor Networks, and Information Processing Conf.*, 2005, pp. 169–174.

[17] 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.

[18] L. Mottola and G. P. Picco, "Programming wireless sensor networks with logical neighborhoods," in *Proc. of the 1st International Conference on Integrated Internet Ad Hoc and Sensor Networks*, 2006.

[19] S. Kabadayi and C. Julien, "A local data abstraction and communication paradigm for pervasive computing," in *Proc. of the 5th Annual IEEE International Conference on Pervasive Computing and Communications*, 2007, (to appear).

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

[21] J. Hill, R. Szewczyk, A. Woo, S. Hollar, D. Culler, and K. Pister, "System architecture directions for networked sensors," *ACM SIGOPS Operating Systems Review*, vol. 34, no. 5, pp. 93–104, December 2000.

[22] 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*, 2002, pp. 85–95.

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